Pandas: Robust Locality-Aware Scheduling With Stochastic Delay Optimality

Qiaomin Xie, Mayank Pundir, Yi Lu, Cristina L. Abad, and Roy H. Campbell, *Life Fellow, IEEE*

***Abstract*— Data locality is a fundamental problem to data-parallel applications where data-processing tasks consume different amounts of time and resources at different locations. The problem is especially prominent under stressed conditions such as hot spots. While replication based on data popularity relieves hot spots due to contention for a single file, hot spots caused by skewed *node popularity*, due to contention for files *co-located* with each other, are more complex, unpredictable, hence more difficult to deal with. We propose Pandas, a light-weight acceleration engine for data-processing tasks that is robust to changes in load and skewness in node popularity. Pandas is a stochastic delay-optimal algorithm. Trace-driven experiments on Hadoop show that Pandas accelerates the data-processing phase of jobs by 11 times with hot spots and 2.4 times without hot spots over existing schedulers. When the difference in processing times due to location is large, such as applicable to the case of memory-locality, the acceleration by Pandas is 22 times.**

***Index Terms*— Data processing systems, MapReduce, locality-aware scheduling, hot-spot mitigation.**

I. Introduction

DATA-PARALLEL applications have become widely popular for processing large data sets from online social networks, search engines, scientific research and health-care industry. MapReduce [13] pioneered the model, while systems like Dryad [19] and Map-Reduce-Merge [39] generalized the types of data flow.

Data files stored on distributed file systems like HDFS are divided into data blocks of fixed size. Each data block has three replicas that are placed on random nodes. Each job is broken into tasks. For instance, there is one map task per data block for MapReduce jobs. Data locality is a fundamental problem to all data-parallel applications where data-processing tasks consume different amounts of time and resources at different locations. Many locality-aware scheduling algorithms have been proposed [17], [18], [20], [21], [43].

Manuscript received July 4, 2015; revised February 19, 2016 and July 4, 2016; accepted August 3, 2016; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Mellia. A preliminary version of this paper focusing on mathematical proofs appeared as a conference publication [38].

Q. Xie and Y. Lu are with the Electrical and Computer Engineering Department, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (e-mail: qxie3@illinois.edu; yilu4@illinois.edu).

M. Pundir and R. H. Campbell are with the Computer Science Department, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (e-mail: pundir2@illinois.edu; rhc@illinois.edu).

C. L. Abad is with the Computer Science Department, Escuela Superior Politécnica del Litoral, Guayaquil 090150, Ecuador (e-mail: cabad@fiec.espol.edu.ec).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2016.2606900

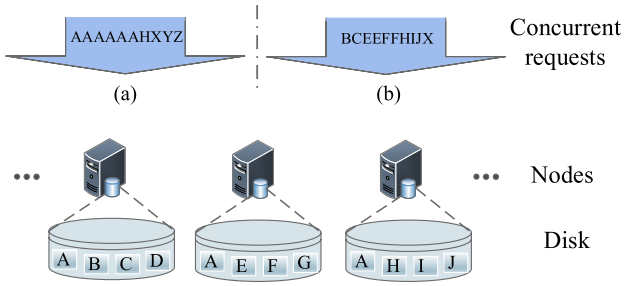


Fig. 1. (a) A hot-spot consisting of 3 nodes, caused by contention for a single data block. Each data block is represented by a distinct alphabet. (b) Same hot-spot caused by contention for data blocks co-located on the respective nodes. (Assume each node can only process one task.)

However, they are not robust to changes in load or skewness of data popularity [5]. As a result, hot-spots form and become bottlenecks in the system. As we observed in our experiments, hot-spots have a prolonged effect on job completion times even after they disappear as delay is propagated due to inefficient scheduling.

Replication based on data popularity [5] can alleviate hot-spots due to contention for popular data blocks, and achieves a 20.2% reduction of the job completion times of Hadoop jobs. However, hot-spots caused by skewed *node popularity*, due to contention for data blocks *co-located* with each other, are more complex, unpredictable, hence more difficult to deal with. Figure 1(a) illustrates a hot-spot caused by contention for the highly popular data block A. Figure 1(b) illustrates a hot-spot caused by contention for different data blocks co-located on the three nodes, assuming that other nodes containing replicas of requested data blocks are busy serving other blocks. Each data block in Figure 1(b) does not have a large number of concurrent accesses, hence will not have extra replicas under Scarlett [5], but collectively they cause contention at their respective nodes, resulting in a hot-spot.

Workload studies [4], [5] have shown that the skewed data popularity in MapReduce clusters is not as dramatic as in other contexts (e.g., web and peer-to-peer content). Instead of a few extremely popular files, there are a large number of moderately popular files. In fact, only 1.5% of all files, or 18% of data, have at least four concurrent accesses [5]. However, less popular files still contribute to skewed node popularity, which occurs in an unpredictable manner due to the random data placement employed by distributed file systems [14], [28]. Hot-spots caused by skewed node popularity are more difficult to detect before they cause prolonged delays, as it is not

known a priori which replica of a data block will be processed. Moreover, hot-spots shift as data popularity experiences even a gradual change. We defer a full exposition of the problem of hot-spots to Section III.

We propose Pandas, a light-weight acceleration engine for data-processing tasks that is robust to changes in load and skewness in node popularity. Pandas is broadly applicable to all data-parallel applications. It works with job-level fairness [43], straggler mitigation [6], [7] and schedules that optimize makespans [22], [36], [44], as these job-level algorithms specify the priority among different jobs or phases of a job whereas Pandas specifies the priority among tasks of any data-processing phase.

Pandas is based on a recently proposed stochastic delay-optimal algorithm for affinity scheduling [37]. Consequently Pandas provides provably optimal performance in a stochastic environment, with no assumption on node popularity or load. Pandas consists of two main steps:

1) *Early detection of hot-spots:* While early detection is highly desirable for relieving hot-spots, it is not straightforward. As each data block has multiple replicas, it is incorrect to estimate the traffic at a node by simply summing all workload whose data reside on this node. Pandas accurately detects a hot-spot before it causes excessive delay by monitoring a queuing structure with appropriate load balancing.
2) *Serve the right remote task:* Timely service of remote tasks, that is, tasks whose data need to be fetched over the network, by lightly loaded nodes helps relieving hot-spots. However, not all tasks are equal. Pandas ensures that only tasks contributing to potential hot-spots are served remotely.

We have integrated Pandas with the Hadoop FIFO scheduler and Fair scheduler (HFS). Each scheduler retains its original job priority. To focus on the performance benefit brought by Pandas to the data-processing phase, we use the SWIM workload [12] to obtain realistic characteristics of data-processing tasks, but with empty reduce phases, as the time taken by the reduce phase can be orthogonally improved by other techniques [16], [29]. The workload study [11] that SWIM is based on also shows that 75% of jobs in the Facebook trace have no shuffle stage and the map outputs are directly written to the file system.

We evaluate Pandas in a variety of environments including Amazon's Elastic Compute Cloud (EC2), a private cluster and via large-scale simulations. Pandas-accelerated FIFO scheduler achieves 11-fold improvement in average job completion time with hot-spot and 2.4-fold improvement without hot-spot over the Hadoop FIFO scheduler. When the difference in processing times due to location is large such as applicable to the case of memory-locality, Pandas-accelerated Fair scheduler achieves 22-fold improvement over HFS during a hot-spot.

II. Related Work

Locality-Aware Scheduling: Among the existing locality-aware scheduling algorithms, the work most closely-related to ours is delay scheduling in HFS [43] and JSQ-MaxWeight [35]. HFS focuses on the conflict between data locality and fairness among jobs. While fairness is a job-level priority, delay scheduling is a task-level algorithm that specifies the priority among map tasks based on their data location. However, delay scheduling makes assumptions that may not hold universally: (a) task durations are short and bimodal, and (b) a fixed waiting time parameter works for all loads and skewness of traffic. These assumptions make it difficult for delay scheduling to adapt to changes in workload, network conditions, or node popularity. In contrast, Pandas makes no assumption on task durations and is provably robust to the aforementioned changes. Pandas is readily integrated with the fairness part of HFS, as demonstrated in this paper.

The JSQ-MaxWeight algorithm [35] is shown to be throughput-optimal, but only heavy-traffic optimal for a very special case where a node is either overloaded or has zero traffic. For all other cases, it can be shown that JSQ-MaxWeight is not heavy-traffic optimal. In contrast, the affinity-scheduling algorithm, which Pandas is based on, studies the same stochastic model as JSQ-MaxWeight and has been shown to be heavy-traffic optimal for all traffic scenarios [38]. In fact, it is the only known heavy-traffic optimal algorithm for this model. This stochastic delay optimality enables Pandas to be robust to changes in load and node popularity, while always achieving fast completion times. It has been shown via simulation that our affinity-scheduling algorithm achieves 4-fold improvement in average task completion time over JSQ-MaxWeight [38].

Other locality-aware algorithms include Quincy [20], Bar [21], Maestro [18] and Matchmaking [17]. Like HFS, Quincy [20] has a task-level algorithm that works with the fairness job priority. In particular, at each task arrival and departure, Quincy solves a min-cost flow problem that optimizes a linear combination of data bytes transferred, penalty of an unscheduled task and penalty of killing a running task. The optimization is greedy at each step, does not consider the stochastic arrivals of jobs, and does not translate into optimal job completion times over a long horizon. Bar [21] assumes that all jobs have the *same* task execution time in its optimization of makespan. Maestro [18] assumes the knowledge of the number of data blocks on each node *to be processed* in the future, and assumes that each data block is processed exactly once. Matchmaking [17] avoids tuning the waiting time parameter of HFS by making each node wait exactly one heartbeat interval before acquiring a remote task. However, this fixed waiting time still makes it difficult to adapt to skewed node popularity and varying loads.

In addition, there is work focusing on locality and virtual machines (VMs). The ILA scheduler [9] adds a new level of locality due to co-locating VMs on the same node and makes the waiting time of delay scheduling [43] proportional to the data size, which can be smaller than the maximum data block size of 128 MB. While it is not clear whether ILA's setting of parameter is optimal, Pandas can be readily extended to include different levels of locality. Purlieus [24] couples data placement and VM placement, but does not consider task assignment, hence is complementary to Pandas.

Data Placement: Several data placement techniques have been used to improve locality. Scarlett [5] adopts a proactive

replication scheme that periodically replicates files based on predicted data popularity. It focuses on data that receives at least three concurrent accesses. However, it does not consider *node popularity* caused by co-location of moderately popular data, which is the focus of this paper. DARE [3] adopts a reactive approach that probabilistically retains remotely retrieved data and evicts aged replicas. Its reactive nature makes its performance depend on appropriate and timely remote services. As Pandas serves the *right* remote tasks, it will be a valuable complement to DARE. PACMan [8] caches data in memory to improve job completion times. Pandas can be readily extended to include memory locality in scheduling to reap the benefit of cached data.

Scheduling MapReduce Jobs: There has been much recent work on scheduling MapReduce jobs, including improvement of shuffle phase [23], [34], [40], joint scheduling of shuffle and reduce phase [16], [29], joint scheduling of map and reduce phase [22], [30], straggler mitigation [6], [7], [27], and optimization of job schedules to minimize average response time and makespan [10], [25], [32], [36], [44]. All these algorithms do not consider locality of map tasks, hence are orthogonal to Pandas. Pandas is designed to work with various job-level and phase-level priorities by assigning the *optimal* data-processing task when the job-level algorithm wants *a* data-processing task assigned.

Resource Packing: Tetris [15] uses multi-resource packing to improve utilization and job completion time. While Pandas is evaluated in a slot-based environment in this paper due to an existing problem with HFS in Yarn (Section VI), it can be extended to work in a multi-resource environment and will be complementary to Tetris.

Application-Specific Scheduling: KMN [31] improves data locality of jobs using a sampled subset of their data. The availability of multiple choices allows it to avoid localized hot-spots in the system. However, as hot-spots caused by node popularity can propagate to a significant fraction of the system due to replicas sharing a workload, as illustrated in Section III, Pandas will be complementary to KMN during the propagation of hot-spots.

III. Problem of Hot-Spots

When the number of concurrent tasks requesting data blocks from a single node exceeds the capacity of the node, *contention* occurs, resulting in hot-spots. Tasks that cannot be accommodated immediately are either queued to wait for local slots or processed on another node that does not store the requested data.

When many concurrent tasks request the same data block that is exceedingly popular and cause contention, we call it a hot-spot due to skewed *data popularity*. Figure 1(a) illustrates such a hot-spot caused by contention for data block A. When concurrent tasks request different data blocks, but these data blocks happen to be co-located on the same node due to the uniformly random data placement by the file system, contention still occurs on this node. Figure 1(b) illustrates a hot-spot due to a highly popular node, where the tasks causing contention request completely different data blocks.

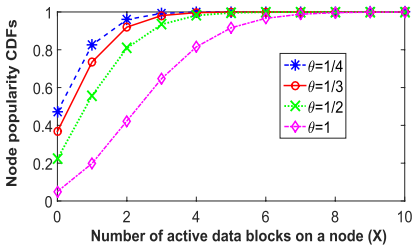


Fig. 2. Node popularity CDFs under the uniformly random data placement assumption for various values of the number of active data blocks ($B = \theta M$).

Of the two types of hot-spots, the former can be alleviated by dynamic replication scheme based on data popularity [5], which, however, cannot avoid the latter. To explore the hot-spot problem caused by files co-located with each other, we first describe the the skewed node popularity problem that arises with uniformly random data placement.

Skewed Node Popularity: Since distributed file systems like HDFS place blocks on random nodes, popular data blocks can co-locate with each other. Consider a $M-$node cluster with one slot per node. Suppose that there are J concurrent jobs that access B distinct data blocks, which we referred to as *active* data blocks. Assume that the cluster is large and $B = \theta M$. Recall that each data block has three replicas that are placed on random nodes. Then for any node, the number of active data blocks it contains (which we denote X) follows Poisson distribution with mean 3θ. A higher value of X implies that this node is more *popular* with multiple active data blocks. Figure 2 shows CDFs of node popularity for different number of distinct data blocks accessed ($B = \theta M$). The higher the value of B, the more skewed to the right the distribution is, i.e., the higher the probability of containing multiple active data blocks. For example, for $B = \frac{1}{3}M$, $\mathbb{P}[X = 0] = \mathbb{P}[X = 1] = 0.37$, $\mathbb{P}[X \geq 2] = 0.26$. Hence a substantial fraction of nodes in the cluster have more than one active data block. Depending on the popularity of data blocks co-located at these nodes, contention can occur on these nodes and result in hot-spots of different types.

Skewed Node Popularity vs. Skewed Data Popularity: These two factors together contribute to the occurrence of hot-spots. A study on MapReduce workload from Bing's Dyrad [5] found that 18% the data have more than three concurrent accesses, and a substantial fraction of the data are less popular, with two or three concurrent accesses (22%). We can assume that 18% of the B data blocks are hot with at least four concurrent tasks, and the remaining 82% data blocks are less popular with at most three concurrent accesses. That is, there are $0.54B$ hot data blocks and $2.46B$ less popular data blocks. Then for any node, the probability that it has at least one hot data block is

$$P_h = 1 - (1 - \frac{1}{M})^{0.54B} \approx 1 - e^{-0.54\theta}.$$

The event that at least two less popular data blocks co-locate on the node, but without any hot data block, happens with

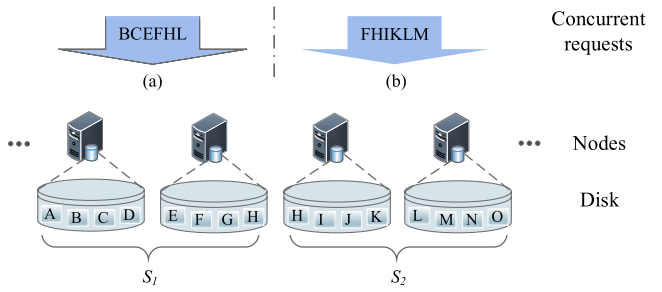


Fig. 3. Hot-spots shift with 50% change in requested data. (a) Concurrent requests create a hot-spot S_1. (b) The hot-spot shifts to S_2 with a 50% change in requested data.

probability

$$P_c = (1-\frac{1}{M})^{0.54B}\left[1-(1-\frac{1}{M})^{2.46B} - \binom{2.46B}{1}\cdot\frac{1}{M}\cdot(1-\frac{1}{M})^{2.46B-1}\right] \approx e^{-0.54\theta}\left[1-e^{-2.46\theta}(1+2.46\theta)\right].$$

For example, for $\theta = \frac{1}{3}$, $P_h \approx 0.16$, $P_c \approx 0.17$. Hence about 16% of nodes will become hot-spots caused by contention for exceptionally popular data blocks, as illustrated in Fig. 1(a). We call it a hot-spot due to skewed *data popularity*. On the other hand, the co-location of less popular files contribute to a significant fraction of nodes with at least two active data blocks. Contention can occur on these nodes due to concurrent access of different data blocks, as illustrated in Fig. 1(b). We call it a hot-spot due to skewed *node popularity*.

Scarlett [5] sets the replication factor of each file proportional to its maximum number of concurrent accesses. Thus contentions caused by exceptionally files can be alleviated under Scarlett. However, since less popular files do not have extra replicas, the skewed node popularity caused by less popular files continues to cause contention.

Hot-Spots Shift: Data popularity changes gradually over time [4], [5]. While an exceedingly popular file will survive a gradual change and remain popular, moderately popular files may not receive an access for a significant period of time. Studies of data popularity [5] show that approximately 50% of the files accessed in a particular hour are not accessed in the preceding or succeeding hour, and temporal locality continues to decay as the number of hours increases. Figure 3 illustrates a hot-spot shifting due to change in file popularity. Assume that each node can only process one task, and nodes in S_1 and S_2 are idle. Figure 3(a) shows that concurrent requests for data blocks B, C, E and F create contention on nodes in S_1, assuming that other nodes storing replicas of requested data blocks are busy processing other data blocks. Figure 3(b) illustrates a 50% change in requested data blocks, with data blocks B, C and E replaced by I, K and M. As a result, contention no longer exists on the nodes in S_1. Instead, the hot-spot has shifted to the nodes in S_2. This makes prediction of hot-spots due to skewed node popularity difficult.

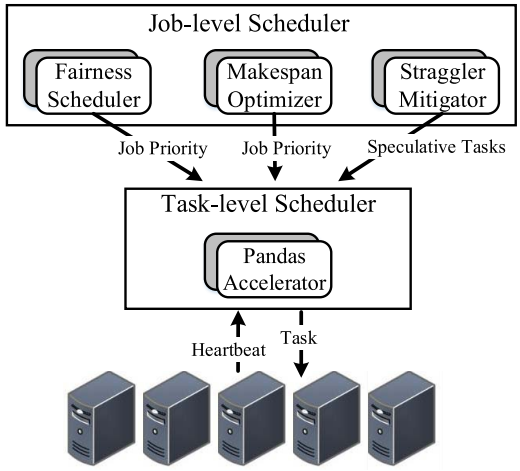


Fig. 4. Scheduler design with Pandas acceleration engine.

IV. Pandas System Design

Pandas (Priority Algorithm for Near-Data Scheduling) is a *task-level* algorithm that specifies the priority among tasks of any data-processing phase by considering data locality. A job-level algorithm, on the other hand, specifies the priority among different jobs or phases of a job, and considers all tasks in the same phase to be the same. Pandas is encapsulated in a light-weight acceleration engine for data-processing tasks that readily take job priority information and straggler tasks from job-level algorithms. Figure 4 illustrates the information flow between Pandas and job-level algorithms such as fairness, makespan optimization and straggler mitigation. The details of task initialization on a node are handled by existing modules in the framework.

Pandas detects hot-spots early by estimating the expected amount of contention at a node. It maintains a queuing structure to identify a subset of nodes that have the potential to become hot-spots via load balancing. Section IV-A describes the details of hot-spot detection. The identification of potential hot-spots allows appropriate decisions of whether to serve a remote task and which remote task to serve. This enables timely relief of hot-spots without assigning too many remote tasks and sacrificing system throughput. Section IV-B describes the details of task assignment.

A. Early Detection of Hot-Spots

While it is easy to detect a highly popular file by simply counting the number of requests for this file, detecting a highly popular *node* is less straightforward. As each data block has multiple replicas and task processing time varies, it is not known a priori where a task should be processed. We will use the following definitions: For each task, we call a node a *local node* for the task if the data block to be processed by the task is stored locally, and we call this task a *local task* for the node. Analogously, we call a node a *rack-local node* if the data block to be processed by the task is not stored on the node, but in the same rack as the node, and we call this task a *rack-local task* for the node. A node is a *remote node*

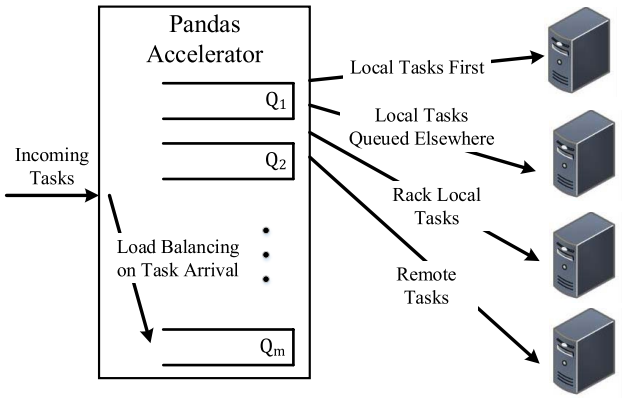


Fig. 5. The Pandas acceleration engine.

if it is neither local nor rack-local for the task and this task is called a *remote task* for the node.

Queuing Structure: Pandas estimates the number of concurrent requests at each node using a queuing structure within the scheduler as illustrated in Figure 5. The queuing structure contains M queues, where M is the number of nodes in the cluster. The m-th queue, denoted by Q_m, buffers the tasks that are *projected* to be processed at node m based on Pandas' estimation. Note that some of these tasks might not be processed at node m eventually: A task can be processed by another node that also hosts its data as a local task, or be processed as a rack-local or remote task. For node m, we call tasks local to it but buffered at Q_n, $n \neq m$, where node n is another local node for the tasks, *local tasks queued elsewhere*. In implementation, the queues need not contain actual tasks. Since Pandas only estimates the number of concurrent accesses and leaves the task initialization details to other modules, pointers to tasks will equally suffice.

Load Balancing: When a job arrives, Pandas performs *load balancing* on all its tasks. For each task, Pandas compares the lengths of its local queues, and inserts the task into the shortest queue. Ties are broken randomly. Load balancing is an estimate of where each task should be processed based on the *expected* amount of load on each node, since the exact processing time of each task is unknown. The purpose of load balancing is not to permanently assign a task to a node. Rather the purpose is to push away concurrent tasks from a potential hot-spot to other local nodes with less contention, if such nodes exist.

Load balancing reveals two distinct subsets of nodes. One subset do not suffer contention and will help relieve hot-spots by serving rack-local and remote tasks. The other subset suffer contention and constitute potential hot-spots. These nodes might require rack-local and remote services, the decision of which is the subject of Section IV-B.

B. Serve the Right Remote Task

After potential hot-spots are identified, Pandas decides on the sequence of task assignment as follows:

Local Tasks First: When node m becomes idle, the scheduler sends the task with the highest job priority from Q_m.

Local Tasks Queued Elsewhere: If Q_m is empty, the scheduler searches for other tasks local to node m in the system. Let $\mathcal{L}_t$ be the set of queues with tasks local to node m. The scheduler sends a local task to node m from the longest queue in the set $\mathcal{L}_t$. The service of local tasks queued elsewhere is a compensation for errors in the estimation stage: A task expects to be served the fastest at local node n, hence joins Q_n, but another of its local nodes, m, turns out to be less loaded than expected, and finishes all local tasks buffered at the local queue Q_m before the task is served.

Rack-Local Tasks: When node m becomes idle and there are no local tasks in the system, the scheduler sends a rack-local task to node m from the longest queue within the same rack, Q_r^{max}, if its queue length $|Q_r^{max}|$ exceeds a predetermined threshold T_r.

Remote Tasks: When node m becomes idle and there are neither local nor rack-local tasks in the system, the scheduler sends a remote task to node m from the longest queue in the system, Q_s^{max}, if its queue length $|Q_s^{max}|$ exceeds another predetermined threshold T_s.

The thresholds T_r and T_s are used to determine whether the task in question should stay in its local queue. They are computed as follows. Let σ_l be the average processing time (not including waiting time) of a local task, σ_r be that of a rack-local task, and σ_s be that of a remote task. Then,

$$T_r = \frac{\sigma_r}{\sigma_l}, \quad T_s = \frac{\sigma_s}{\sigma_l}. \tag{1}$$

Note that σ_s is the average processing time of *all* remote tasks in the system, so are σ_l and σ_r the average processing times of the respective tasks, hence they are not location-dependent and easy to compute. We refer to $\frac{\sigma_r}{\sigma_l}$ and $\frac{\sigma_s}{\sigma_l}$ as the *average slowdown* in the rack and in the system respectively.

When the queue length $|Q_s^{max}|$ exceeds T_s, the average waiting time of the last task in Q_s^{max} is $|Q_s^{max}|\sigma_l > T_s\sigma_l$, but the average processing time of the task on an idle remote node is $\sigma_s = T_s\sigma_l$, hence the task benefits from being processed remotely. The same argument applies to the assignment of rack-local tasks, with $|Q_r^{max}|$ replacing $|Q_s^{max}|$ and T_r replacing T_s.

Remark: When a system consists of only one rack, only the remote threshold T_s is used for the longest queue in the system. When a system has an unknown rack structure, as in our experiments on AWS, Pandas regards all queues as remote and only the remote threshold T_s is used for the longest queue in the system.

In our experiments, we found that the average slowdown changes with load and traffic distributions. Although we use a fixed threshold for each experiment, the performance will likely be improved even further if we dynamically set the threshold based on the average slowdown measured over a sliding window. We leave dynamic thresholds for future work. In our sensitivity analysis in Section VII, we show that a small deviation from the correct threshold does not affect the performance of Pandas.

Pandas Extension: The Pandas algorithm is based on a versatile model that enables it to be extended beyond map-reduce clusters. Tasks with data at different locations are modeled using different processing rates that depend on the locality of the data. In particular, the Pandas algorithm can

be readily extended to systems with multi-level locality where the average task processing rate depends on the corresponding data location, whether it is in memory, on local disks, in a local rack, in the same cluster or in a different data center. In particular, for a system with geo-distributed data [26], [33], the data residing in another data center will be processed by tasks of the slowest processing rate, reflecting the potential variation in network transfer when retrieving data from a different data center.

V. Analysis

To analyze the performance of a scheduler, the two essential criteria are throughput and delay: Throughput is equivalent to the efficiency and robustness of the system, and delay is equivalent to the completion time of tasks. Any other criterion, such as data locality or a cost function involving data transfer and waiting time, is meaningful only when translated into long-term throughput and delay in a stochastic environment.

The challenge of analyzing a scheduler lies in the stochastic nature of job arrivals and completion. There is randomness in the location of the data chunk for an incoming task, and also in the processing time of a task. We use stochastic analysis to study the performance of Pandas in equilibrium. Although equilibrium is never strictly achieved in reality, it usually well approximates the system performance after it runs for a reasonable amount of time, such as a few hours.

We analyze a basic version of the Pandas algorithm, where there are no rack-local tasks, and the algorithm does not assign local tasks queued elsewhere. We assume that processing times follows exponential distributions. The performance of the basic version is strictly worse than Pandas, hence any property of the basic algorithm will hold for Pandas. Before we state the theorems, we need a few definitions:

Load Vector: As the processing rates depend on the task-node pair, we index a task type by its three local nodes. A load vector is a vector consisting of the load for each task type.

Capacity region. All possible load vectors form a continuous region in a high-dimensional space. If there exists an algorithm such that a particular load vector is stabilizable, that is, the mean completion time is finite, we say this load vector is within the capacity region.

Theorem 1 (Throughput Optimality): Pandas is throughput optimal. That is, it stabilizes any load vector strictly within the capacity region.

Throughput optimality means that an algorithm is robust so that whichever load vector strictly within the capacity region is imposed, the system is efficient enough to achieve a finite task completion time. According to Little's Law, this implies that the number of tasks queueing in the system is finite, i.e.,

$$\limsup_{t\to\infty} \mathbb{E}\left[\sum_m Q_m(t)\right] < \infty.$$

A drastic increase in completion time indicates that the load vector is close to an algorithm's stability region. We compare the stability region of Pandas against existing schedulers using simulation in Section VII, which shows that the system is stable under Pandas for all loads up to capacity. However, throughput optimality does not guarantee the task completion time to be small.

It is often difficult to analyze delay, or task completion time, in a stochastic environment. One way to assess whether an algorithm is competent is to look at the heavy-traffic regime, where the load vector approaches the boundary of the capacity region and the system becomes critically loaded. In our case, it is possible to show state-space collapse in the heavy-traffic regime, i.e., the queue length vector lives in a one-dimensional space, which enables us to show optimality.

Theorem 2 (Heavy-Traffic Optimality): Pandas is heavy-traffic optimal. That is, it achieves a mean delay that coincides with that of a system where all the servers are pooled together to form a giant server, which is the best achievable with the same amount of resource.

The proof outlines are available in [38], with the complete proofs available in [37]. Note that the theorems only concern task completion times and do not provide guarantees on job completion times. It is in general difficult to analyze job completion times as job completion depends on the completion of the last task, hence makes the state space highly complicated. We study job completion times in various environments in Section VII. In particular, we use simulation to evaluate the performance of Pandas at high load regions, where experiments are not possible. Our evaluation shows that Pandas produces low completion time, for both even and skewed data popularity, confirming the delay optimality of Pandas in heavy-traffic regime.

VI. Case Study: Implementation on Hadoop

Hadoop implements the MapReduce paradigm [13] on top of the Hadoop Distributed File System (HDFS) [28]. HDFS stores data files as blocks of configurable size. Each data block is replicated 3 times by default. A Hadoop job consists of two types of tasks: Map tasks and reduce tasks. A map task processes an input data block and outputs intermediate key-value pairs; a reduce task processes the intermediate files to produce the final output. Each node in a Hadoop cluster sends heartbeats to a centralized scheduler, which assigns tasks to the nodes according to the scheduling policy.

The Hadoop FIFO Scheduler assigns tasks according to the order in which they arrive. It maintains a central queue of jobs. On receiving a heartbeat, the scheduler assigns a task from the head-of-line (HOL) job in the order of local, rack-local, and remote tasks.

The Hadoop Fair Scheduler (HFS) gives priority to jobs based on fairness, i.e., the difference between the fair resource share and the current resource usage. It uses delay scheduling at the task level to relax job priority for an increase in data locality: A job has to wait for more than *nodeLocalityDelay* milliseconds (ms) before it can assign a rack-local task, and another *rackLocalityDelay* ms before it can assign a remote task.

A. YARN Limitations

YARN is the resource manager for Hadoop2. We have incorporated Pandas in Hadoop1 instead of YARN as the implementation of HFS is sub-optimal in the YARN architecture. Note

that YARN does not provide additional performance gain over HFS and the delay scheduling algorithm in HFS is the state-of-art task-level scheduling algorithm. The performance of HFS in Hadoop1 is superior to that in Hadoop2 due to the well-known implementation issues [41], [42], which we explain as follows.

In the YARN architecture, an Application Master (AM) is responsible for requesting resources for a job from a centralized Resource Manager (RM). Each AM to RM heartbeat consists of a list of resource requests for outstanding tasks. Each request contains a container count, a resource vector, and a desired location (a node, a rack, or "$*$" representing any location). For example, for a map task having data on nodes n_1, n_2 and n_3, the AM would issue up to seven requests: one for each node, one for each rack where the nodes reside, and one for $*$, indicating any node. Requests from different tasks for the same node are combined by the AM.

The RM first attempts to satisfy node requests, then rack requests, and lastly $*$ requests. For simplicity of the protocol (payload length and node statelessness), these requests are not equipped with a task ID. When node-local containers are allocated, the RM prunes redundant requests by decrementing the number of rack-local and $*$ requests. However, when the scheduler allocates a container to a rack request, its corresponding node requests are left in the request list as the RM cannot know which requests have become redundant.

This protocol leads to several problems for HFS: (1) The scheduler uses data locality to decide if and for how long an idle node should wait. The redundant requests lead to inaccurate information and throw off the delay scheduling algorithm. (2) The scheduler has no control over which task is actually assigned to a node. It could mistakenly believe that a node-local assignment has been made, when in reality the request was redundant and the AM assigned a non-local task to the node. (3) Since there is no way to eliminate all redundant requests, assignments are made for them. These assignments are released by the AM in the next heartbeat leading to resource under-utilization.

B. Data Structures in Hadoop

We use the following data structures in our implementation of Pandas acceleration engine in Hadoop:

nodeQueues: An array that maintains a per-node queue of local tasks assigned to the node (see queues in Figure 5). It is indexed by the node ID for quick access.

nodeAssigned: It specifies the node to which the task is assigned upon arrival (field added to Hadoop's *TaskInProgress* object).

nodeTasks: An array that maintains a per-node map of *all* tasks local to the node. It provides quick access to local tasks queued elsewhere. Each task is present once in *nodeQueues* but R times in *nodeTasks*, where R is the replication factor of the task's input block.

rackNodes: An array that maintains a per-rack list of nodes on that rack. It is indexed by rack ID and provides quick access to rack-local queues for a given node.

Algorithm 1 Job Arrival in Pandas-Accelerated Schedulers: onJobArrival

Input: Job job

```
1: shortestQueueNode← ∅
2: for task in job do
3:   for node in data-local nodes of task do
4:     if queue of node is shorter than current shortest then
5:       shortestQueueNode← node
6:     end if
7:     Insert task in task map of node in nodeTasks
8:   end for
9:   Insert task in queue of shortestQueueNode in nodeQueues
10:  Set nodeAssigned of task to shortestQueueNode
11: end for
```

Algorithm 2 Pandas-Accelerated FIFO Scheduler: onHeartbeatArrival

Input: Node m

```
1: if queue, q of m in nodeQueues is not empty then
2:   task← head-of-line task from q
3: else if task map, p of m in nodeTasks is not empty then
4:   iterator← iterator over task map, p
5:   while t in iterator do
6:     if queue, q of t.nodeAssigned is longer than current longest then
7:       task← t
8:     end if
9:   end while
10: else if length of longest rack queue, q_r is greater than rack threshold then
11:   task← head-of-line task from q_r
12: else if length of longest remote queue, q_s is greater than remote threshold then
13:   task← head-of-line task from q_s
14: end if
15: schedule task
16: remove task from its queue in nodeQueues
17: remove task from its maps in nodeTasks
```

C. Load Balancing on Job Arrival

The Pandas-accelerated FIFO and Fair Schedulers perform the load balancing operation on job arrival shown in Algorithm 1. For each task in a new job, they check the lengths of the queues on each of the R local nodes for the task, where R is the replication factor of the input data block (lines 3-8). They insert the task in the shortest node-local queue in *nodeQueues*. While iterating over the data local nodes, they insert the task in each node's task map (line 7). They also set the *nodeAssigned* field of the task to the task's shortest node-local queue (line 10).

D. Pandas-Accelerated FIFO Scheduler

The pseudo-code of the Pandas-accelerated FIFO Scheduler is shown in Algorithm 2. The job priority is implicit in the

queuing structure, as each queue maintains a FIFO order. Hence no job priority information is needed by Pandas.

Local Tasks First: On receiving a heartbeat from a node m, the scheduler checks if there are tasks queued in the local queue of m in *nodeQueues* (lines 1-2). If the local queue is not empty, it removes and schedules its head-of-line task. It also removes the task from its R maps in *nodeTasks*.

Local Tasks Queued Elsewhere: If the local queue of the node m is empty, the scheduler looks for local tasks queued elsewhere (lines 3-9). It iterates over the set of tasks in m's map in *nodeTasks* (those local to the node). For each task, it checks the length of the queue in *nodeQueues* to which the task was assigned upon arrival (given by *nodeAssigned*). It schedules the task assigned to the longest queue. After scheduling, it removes the task from *task.nodeAssigned* queue in *nodeQueues* and from its R maps in *nodeTasks*.

Rack-Local Tasks: If there are no local tasks in the system, the scheduler looks for rack-local tasks (lines 10-11). It iterates over the queues of nodes in the rack of node m (determined using *rackNodes*) and picks the longest queue, q_r. If the length of this rack-local queue is greater than the rack threshold, it removes and schedules the head-of-line task from the queue. It also removes the task from its R maps in *nodeTasks*.

Remote Tasks: If there are no local tasks in the system and all rack-local queues for m are shorter than the rack-threshold, the scheduler looks for remote tasks (lines 12-13). It iterates over the node queues outside the rack of node m and picks the longest queue, q_s. If the length of this remote queue is greater than the remote threshold, it removes and schedules the head-of-line task from the queue. It also removes the task from its R maps in *nodeTasks*.

E. Pandas-Accelerated Fair Scheduler

The pseudo-code of the Pandas-accelerated Fair Scheduler is shown in Algorithm 3. Pandas retrieves the fairness job priority from HFS, and applies it to the subset of jobs present *in each queue*. Analogous to the Pandas-accelerated FIFO Scheduler, lines 2-4 implement "local tasks first", lines 5-13 implement "local tasks queued elsewhere", lines 14-16 implement "rack-local tasks", and lines 17-19 implement "remote tasks".

Algorithm 3 Pandas-Accelerated Fair Scheduler - onHeartbeatArrival

Input: Node m

1: $jobs \leftarrow$ candidate jobs from highest priority pool
2: **if** queue, q of m in nodeQueues is not empty **then**
3: sort jobs corresponding to tasks in q using fair sharing
4: $task \leftarrow$ task corresponding to HOL job from sorted list
5: **else if** task map, p of m in nodeTasks is not empty **then**
6: $iterator \leftarrow$ iterator over task map, p
7: **while** $task$ in $iterator$ **do**
8: **if** queue, q of $task.nodeAssigned$ is longer than current longest **then**
9: $longestQueue \leftarrow q$
10: **end if**
11: **end while**
12: sort jobs corresponding to tasks in p assigned to $longestQueue$ using fair sharing
13: $task \leftarrow$ task corresponding to HOL job from sorted list
14: **else if** length of longest rack queue, q_r is greater than rack threshold **then**
15: sort jobs corresponding to tasks in q_r using fair sharing
16: $task \leftarrow$ task corresponding to HOL job from sorted list
17: **else if** length of longest remote queue, q_s is greater than remote threshold **then**
18: sort jobs corresponding to tasks in q_s using fair sharing
19: $task \leftarrow$ task corresponding to HOL job from sorted list
20: **end if**
21: schedule $task$
22: remove $task$ from its queue in nodeQueues
23: remove $task$ from its maps in nodeTasks

VII. Evaluation

We evaluate the acceleration by Pandas in this section. We describe the evaluation setup in VII-A, evaluate Pandas-accelerated FIFO scheduler against Hadoop FIFO scheduler in VII-B, and Pandas-accelerated Fair scheduler against HFS in VII-C. The overhead of Pandas is discussed in VII-D.

A. Evaluation Setup

We discuss the environment, trace characteristics, evaluation metrics and Pandas' thresholds setting in this section.

1) Environment: For both the Elastic Compute Cloud (EC2) [1] and a private cluster, we run a modified version of Hadoop-1.2.1, configured with a block size of 256 MB and a replication factor 3. We use 100 "m3.xlarge" instances on EC2 and 28 "m1.xlarge" instances on OpenStack [2] in the private cluster. Table I shows details of the instances.

TABLE I
Types of Instances Used in the Experiments

	Nodes	Memory (GB)	VCPU	Map Slots
EC2	100	15	4	4
Private	28	16	8	4

a) EC2: We use EC2 for evaluation with a long trace characterized by hot-spots occurring and disappearing. As rack structures are not available in EC2, every other node is regarded as a remote node. Only the average remote slowdown σ_s/σ_l is computed, and the average is taken over all nodes in the system. Slowdown is measured by collecting local and remote task completion times under the Hadoop schedulers.

The remote slowdown was measured to be 2 on EC2 instances [43]. In our experiments, the slowdown varies with the placement of assigned VMs, and tends to increase with load and hot-spots, which lead to a higher level of network congestion and disk contention. The largest average slowdown we measured on EC2 is 6. Figure 6 shows the CDF of slowdown, defined as (processing time of a remote task / σ_l), where σ_l is the average processing time of local tasks. Although 75% of the remote tasks experience a slowdown less than 5, some of the remote tasks experience as much as 45x slowdown.

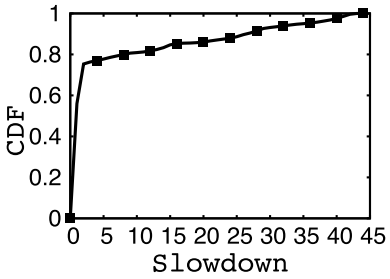


Fig. 6. Slowdown distribution on EC2.

b) Private cluster: We use the private cluster for evaluation under stressed conditions, sensitivity analysis and short traces with fixed load. As all nodes in our private cluster are in the same rack and are exclusively used for the experiment, there is neither traffic from other VMs co-located on the same physical node as in a multi-tenant environment, nor background traffic replicating, redistributing and importing files. As a result, there is virtually no slowdown in the system. For our experiments, we send background traffic from each VM to its neighboring VM to create a slowdown.

c) Large-scale simulation: We use simulation with 500 nodes over a long time horizon to evaluate the performance of Pandas in a large cluster across all loads up to system capacity. It allows us to gain insight into the transient behavior observed with the long trace where hot-spot occurs and disappears. It also allows us to explore the performance of Pandas at load regions where experiments are not possible, as the FIFO scheduler and HFS crash due to excessive queuing of jobs. Pandas is able to avoid excessive queuing due to its throughput optimality.

Our simulation models task processing times as heavy-tailed random variables. We use a truncated Pareto distribution with shape parameter 1.9 to generate the number of tasks for each job. The parameter of HFS is tuned according to HFS evaluation [43] such that 95% data locality is achieved. At each task arrival, a set of three nodes are chosen to be its local nodes. Uniform load is simulated by sampling the nodes uniformly at random. Skewed load is simulated by sampling from half the cluster with probability 0.8, and from the remaining half with probability 0.2. This mimics hot-spots in the presence of uniform traffic. The slowdown of the system is set to 2.

2) Trace Characteristics: We generate traces by sampling jobs from the SWIM benchmark [12] so that 1) we preserve the Pareto job size distribution [43]; 2) The length of the trace and the number of files are appropriately scaled for the capacity of the different clusters by SWIM. We leave the reduce phases empty to focus on the improvement of the data-processing phase only, as the time taken by the reduce phase can be orthogonally improved by other techniques [16], [29].

We did not run the SWIM benchmark directly because SWIM does not model data popularity. It assumes that the file system is only populated with data accessed in this experiment and each task independently chooses its data location. This is different from what happens in a real cluster, where different tasks can access the same data block, hence creating correlated patterns in accesses. SWIM is essentially assuming that each job process a completely different file, which is not the case in practice. Therefore, the data popularity, as well as the node popularity, is always uniform. Moreover, unlike hot-spots caused by skewed data popularity, hot-spots caused by skewed node popularity occur randomly and are hard to reproduce as they depend on the random placement of data blocks by HDFS. A trace constructed with the same distribution of data popularity might cause a hot-spot in one experiment, but not in another.

In view of the above, we generate hot-spots by making a subset of nodes more popular than others. For all experiments on EC2 and the private cluster, hot-spots are generated by increasing the popularity of 40% of nodes in the system, resulting in a skewed node popularity. The actual node popularity varies with the load, which will be specified for each experiment.

Table II shows the job size distribution used for the long trace on EC2, and Table III shows the job size distribution used on the private cluster. They have the same Pareto distribution, but the trace for the private cluster has jobs of smaller sizes as the capacity of the private cluster is smaller.

Table IV shows the long trace on EC2 containing 6 stages with varying load and node popularity, and Table V shows the loads and traffic distributions for the trace on the private cluster. Like the experiments on HFS [43], the inter-arrival time of jobs is generated from an exponential distribution. The uniform node popularity is obtained by using the default

TABLE II
JOB SIZE DISTRIBUTION USED ON EC2

Bins	1	2	3	4	5	6	7
Job Count	570	240	210	120	90	90	60
Map Count per Job	1	2	10	50	100	200	400

TABLE III
JOB SIZE DISTRIBUTION USED ON PRIVATE CLUSTER

Bins	1	2	3	4	5	6	7	8
Job Count	237	95	77	55	42	37	30	27
Map Count per Job	1	2	4	10	25	50	100	200

TABLE IV
TRACE CHARACTERISTICS ON EC2

Job Range	Node popularity, load
1-230	Uniform, 0.24
231-460	Uniform, 0.48
461-690	Uniform, 0.72
691-920	Skewed, 0.48
921-1150	Uniform, 0.48
1151-1380	Skewed, 0.24

TABLE V
TRACE CHARACTERISTICS ON PRIVATE CLUSTER

Job Range	Traffic distribution, load
1-100	Uniform, 0.23
101-200	Uniform, 0.45
201-300	Uniform, 0.68
301-400	Skewed, 0.45
401-500	Uniform, 0.45
501-600	Skewed, 0.23

data placement of SWIM. The skewed node popularity, or hot-spots, are generated by directing all traffic to 40% of the nodes in the system.

3) Evaluation Metrics: We use the following metrics for performance evaluation:

Map Completion Time: It is the average completion time of all map tasks in a trace or in a sliding window. Completion time of a map task is defined as the time interval between task arrival and the moment the task finishes processing. For the long trace, the average is computed in a sliding window of k jobs, i.e., the value at point i on x-axis shows the average computed over jobs $[i-k+1, i-k+2, \ldots i]$ for $i \geq k$ and over jobs $[1, 2, \ldots i]$ for $1 < i < k$. This gives higher resolution to changes in map completion time as hot-spots occur and disappear.

Job Completion Time: It is the average completion time of all jobs in a trace or in a sliding window. Completion time of a job is defined as the time interval between job arrival and the moment the job finishes processing. The sliding window is defined in the same way as for map completion time above.

Data Locality: It is measured as the percentage of map tasks processed at a local node.

Speed-Up: Speed-up in job completion time is defined as

$$\text{Speed-up} = \frac{T - T_P}{T}, \tag{2}$$

where T is the original job completion time under FIFO or HFS, and T_P is that with Pandas acceleration. A negative speed-up indicates that the job is slowed down with Pandas. Note that, with this definition, the maximum possible speed-up is 100%, corresponding to $T_P = 0$, but the maximum possible slowdown is infinite. A 100% slowdown indicates that the job completion time has doubled.

Jobs in Bins: For the detailed analysis on the private cluster, we divide jobs into three classes based on their sizes in Table III: 1) Small jobs (placed in bin $1-3$, having less than 5 map tasks), 2) medium jobs (placed in bin $4-6$, having $10-50$ map tasks) and 3) large jobs (placed in bin $7-8$, having at least 100 map tasks).

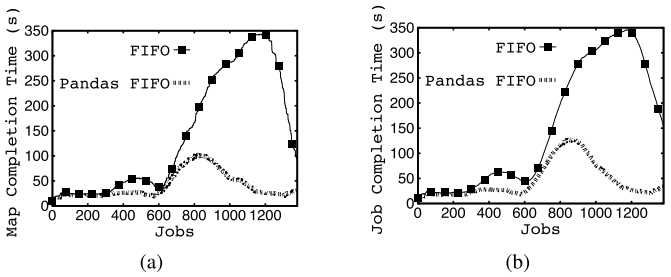
4) Pandas Thresholds: On both EC2 and the private cluster, only the remote threshold T_s is used due to the absence of rack structure. We set T_s based on the slowdown measured with the default Hadoop schedulers. In all experiments, the thresholds are set to fixed values for each trace, even if the average slowdown varies with changing loads and node popularity.

B. Pandas-Accelerated FIFO

1) Long Trace on EC2: We run a trace of 1380 jobs on 5000 files on EC2. The job size distribution is given in Table II. The average slowdown measured with this set of VMs under FIFO is 1.5, hence we set the Pandas remote threshold T_s to 2.

Figure 7 shows the average task completion time and job completion time in a sliding window of 230 jobs. Before hot-spots occur, Pandas-accelerated FIFO outperforms FIFO at all times and the largest improvement of 2.3-fold reduction occurs for jobs $231-460$ at load 0.48.

As soon as the hot-spots occur, the performance of FIFO degrades drastically. With 0.48 skewed load,



(a) (b)

Fig. 7. Pandas-accelerated FIFO achieves up to 11-fold improvement over FIFO on EC2. (a) Task completion time. (b) Job completion time.

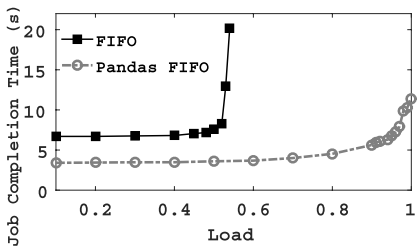


Fig. 8. Average job completion time for uniform load.

the completion time with the FIFO Scheduler severely degrades, while Pandas-accelerated FIFO consistently produces low completion times. Even when the hot-spot disappears, its lingering effect makes the completion time with FIFO continue to increase sharply, while Pandas-accelerated FIFO produces much lower completion time under uniform load. For instance, with jobs $921-1150$, Pandas-accelerated FIFO achieves 11-fold improvement over FIFO. The completion time with FIFO improves gradually with the skewed load of 0.24. In this scenario, the load is skewed but low, hence contention occurs less frequently. The last set of jobs $1151-1380$ receives a 4.5-fold improvement in job completion time with Pandas-accelerated FIFO.

2) Large-Scale Simulation: The behavior of FIFO is clearer with simulation over the entire range of loads within system capacity. Figure 8 shows that with uniform data locality, FIFO incurs very large delay beyond 0.6 load. Figure 9 shows that with hot-spots, FIFO incurs very large delay beyond 0.52 load. FIFO's aggressive assignment of remote tasks and waste of throughput have resulted in instability. This explains the drastic increase in completion time under FIFO in Figure 7. In contrast, Pandas is throughput-optimal and heavy-traffic optimal, hence producing low delays for all loads up to capacity. For both scenarios, the corresponding standard variations are dramatically small (of order 10^{-3}), thus we do not show the error bars on the graph.

3) Detailed Performance on Private Cluster: We run a short trace with the job size distribution in Table III, but scaled to a total of 192 jobs and 3700 map tasks. The trace accesses 1000 files at 0.2 load. The threshold T_s is set to 2. Table VI shows the average map task and job completion time for uniform and skewed loads. Even at such a low load, Pandas-accelerated FIFO achieves 2.38-fold and 2-fold improvements in average job completion time for uniform and skewed loads respectively.

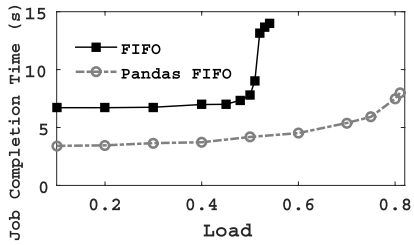
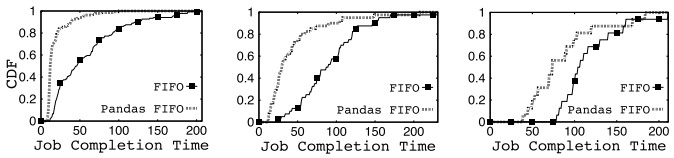


Fig. 9. Average job completion time for skewed load.

TABLE VI
PANDAS-ACCELERATED FIFO OUTPERFORMS FIFO AT 0.2 LOAD

	Average Map Completion Time (s)		Average Job Completion Time (s)	
Workload Behavior	FIFO	Pandas FIFO	FIFO	Pandas FIFO
Uniform	43.37	30.88	70.25	29.48
Skewed	73.44	58.12	143.3	71.86



(a) (b) (c)

Fig. 10. Average job completion time at 0.2 uniform load. (a) Bin 1-3. (b) Bin 4-6. (c) Bin 7-8.

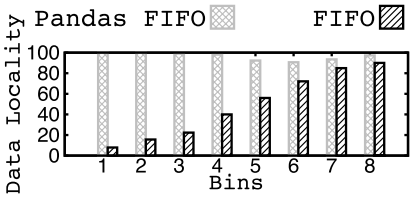


Fig. 11. Data locality at 0.2 uniform load.

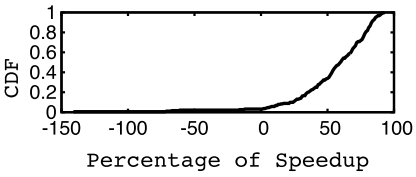


Fig. 12. Speed-up of jobs at 0.2 uniform load.

We focus on uniform load. Similar behavior is observed under skewed load.

Figure 10 shows the CDFs of job completion time for the three classes of jobs. Pandas reduces almost all job completion times and only the largest jobs experience a slight increase in completion time. This is a result of the improvement in overall system efficiency by Pandas.

Figure 11 shows the data locality of FIFO and Pandas-accelerated FIFO for each bin. Pandas-accelerated FIFO achieves almost 100% data locality for all bins. Not surprisingly, the largest improvement is observed for the small jobs as they have the fewest choice of local nodes and are frequently assigned to remote nodes under FIFO. The improvement in system throughput by Pandas also leads to a larger number of idle nodes, hence higher data locality.

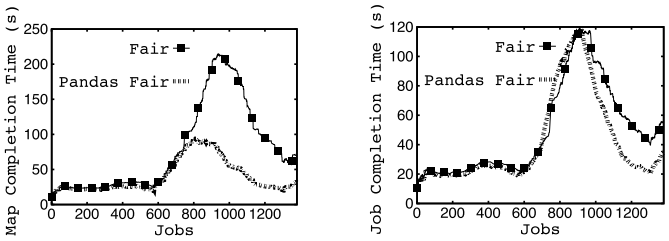
Figure 12 shows the speed-up in job completion times. We observe that most jobs experience a speed-up, with 50.4% of jobs experiencing at least a 60% speed-up, corresponding to 2.5-fold reduction in completion time, and 19.17% of jobs experiencing at least a 80% speed-up, corresponding to 5-fold reduction. Only 3.1% of jobs experience a slowdown, with the largest slowdown being 140%, corresponding to 2.4 times the completion time under FIFO.

4) Sensitivity Analysis: We evaluate the impact of the variation in threshold values on performance. A remote threshold lower than the corresponding slowdown makes the scheduler assign remote tasks too aggressively. On the other hand, a remote threshold higher than the slowdown makes the scheduler too conservative, hence not relieving the hot-spots fast enough.

Table VII shows the average map and job completion time of Pandas-accelerated FIFO with threshold values 3, 5 and 7 when the average slowdown observed is 5. Even with an inaccurate threshold, we observe 9.25-fold improvement over FIFO in average job completion time while we achieve 11-fold improvement with the correct threshold.

TABLE VII
PANDAS-ACCELERATED FIFO WITH DIFFERENT THRESHOLD VALUES AT 0.3 SKEWED LOAD

	FIFO	Threshold $= 3$	Threshold $= 5$	Threshold $= 7$
Map (s)	159.23	42.67	38.17	43.72
Job (s)	381.26	41.2	34.75	38.96



(a) (b)

Fig. 13. Pandas-accelerated Fair achieves up to 47% improvement over HFS. (a) Task completion time. (b) Job completion time.

C. Pandas-Accelerated Fair

1) Long Trace on EC2: We run the same trace as for FIFO in Section VII-B1 with the same threshold $T_s = 2$. Figure 13 shows the average task completion time and job completion time in a sliding window of 230 jobs. Pandas' performance is comparable to HFS under uniform load and during the first hot-spot. However, after the first hot-spot, Pandas accelerates tasks by 45% and jobs by 47%.

With the default waiting time parameter, HFS is sufficiently aggressive in relieving hot-spots at this load although at the cost of wasting throughput and affecting later jobs.

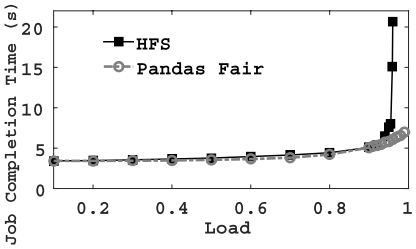


Fig. 14. Average job completion time for uniform load.

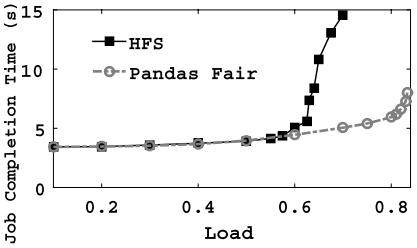
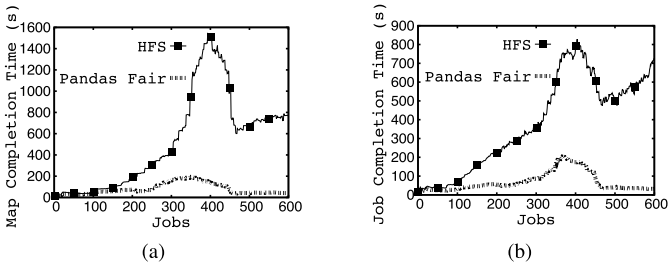
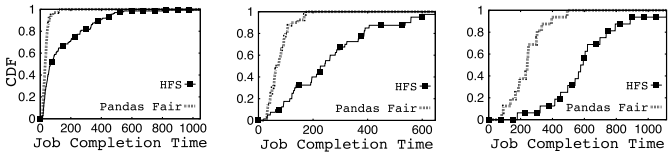


Fig. 15. Average job completion time for skewed load.



(a) (b)

Fig. 16. Pandas-accelerated Fair achieves up to 22-fold improvement over HFS on the private cluster. (a) Task completion time. (b) Job completion time.



(a) (b) (c)

Fig. 17. Average job completion time at 0.68 uniform load. (a) Bin 1-3. (b) Bin 4-6. (c) Bin 7-8.

The acceleration of HFS is not as large as that of FIFO as HFS has a different capacity region and can accommodate a higher load than 0.48. Our experiment at a higher load crashed due to excessive queuing of jobs in the HFS scheduler, although Pandas-accelerated Fair did not suffer from queuing due to its throughput optimality. We explore the behavior of HFS and Pandas acceleration at higher loads using simulation in Section VII-C2.

2) Large-Scale Simulation: Since the performance of HFS depends on the waiting time parameter, we tune the parameter according to HFS evaluation [43] so that HFS achieves 95% data locality. Figure 14 shows that with uniform node popularity, HFS incurs drastic delay only beyond 0.95 load. Figure 15 shows that with hot-spots, HFS incurs high delay beyond 0.65 load. At lower loads, Pandas achieves negligible acceleration, while beyond 0.95 and 0.65 loads respectively, Pandas achieves very large acceleration.

We observe that a high data locality of 95% favors uniform node popularity. When the waiting time parameter is tuned for lower data locality, HFS incurs large delays at a smaller load for uniform, but at a larger load for hot-spots, as in Figure 13. The waiting time parameter yields a trade-off of performance between the two scenarios. But in both cases, HFS has a larger capacity region than FIFO, and large improvement by Pandas will occur only at loads beyond 0.48.

3) A Stressed Test on Private Cluster: We run a trace of 600 jobs on 1000 files on the private cluster. Table III shows the job size distribution while Table IV shows load and node popularity with 100 jobs per segment. The peak slowdown under HFS is 30, which is a stressed scenario with a large amount of network contention, or in an environment where the difference in processing time due to location is large such as with memory-locality. We set $T_s = 30$.

Figure 16 shows the average task completion time and job completion time in a sliding window of 100 jobs. Before hot-spots occur, Pandas-accelerated Fair outperforms HFS at all times and the largest improvement of 4.1-fold reduction in average job completion time occurs for jobs $201 - 300$, even if the average task completion time experiences only a 2.7-fold reduction.

TABLE VIII
PERFORMANCE AT 0.68 LOAD

	Average Map Completion Time (s)		Average Job Completion Time (s)	
Workload Behavior	HFS	Pandas Fair	HFS	Pandas Fair
Uniform	194.9	85.05	209.71	61.93
Skewed	913.99	194.41	610.71	182.4

As the first hot-spot occurs, the improvement in task completion time reaches 11-fold, while that in job completion time remains around 4.5-fold. Unlike FIFO in Figure 7, HFS recovers from the hot-spot much faster, leaving a conspicuous peak in the curve for jobs $301 - 400$. Another peak appears towards the end of the curve as the second hot-spot occurs. The improvements in task and job completion times reach 15 and 12-fold respectively for jobs $401 - 500$, as Pandas-accelerated Fair recovers from the hot-spot. The improvement reaches its maximum of 18 and 22-fold respectively at the second hot-spot.

4) Detailed Performance on Private Cluster: We run the same trace as in Section VII-B3 at 0.68 load. We set T_s to 8 for uniform load and 27 for skewed load based on the average slowdown measured with HFS. Table VIII shows the average map and job completion times. Pandas-accelerated Fair achieves more than 3.3-fold improvement in average job completion time for both uniform and skewed loads.

We focus on uniform load. Similar behavior is observed under skewed load.

Figure 17 shows the CDF of job completion time. Pandas produces significant improvement for all three classes of jobs. Figure 18 shows the data locality of HFS and Pandas-accelerated Fair for each bin. Pandas achieves close to 100%

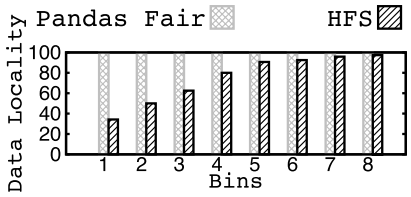


Fig. 18. Data locality at 0.68 uniform load.

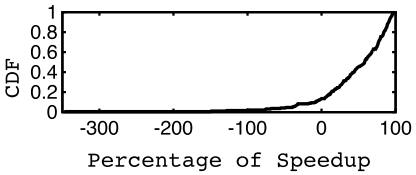


Fig. 19. Speed-up at 0.68 uniform load.

TABLE IX
PANDAS-ACCELERATED FAIR WITH DIFFERENT THRESHOLD VALUES AT 0.6 UNIFORM LOAD

	HFS	Threshold $= 5$	Threshold $= 8$	Threshold $= 10$
Map (s)	194.9	103.34	85.05	115.59
Job (s)	209.71	71.43	61.93	73.74

TABLE X
SCHEDULING DELAY

Scheduler	FIFO	Pandas FIFO	Fair	Pandas Fair
Delay (ms)	0.96	0.75	0.81	1.15

data locality whereas HFS, with its default configuration, achieves only $30 - 60\%$ data locality for the small jobs. We also plot the CDF of speedup for each job in Figure 19. Most jobs experience a speed-up, with 46.87% of jobs experiencing at least a 60% speed-up, corresponding to at least 2.5-fold reduction in completion time, and 24.48% of jobs experiencing at least a 80% speed-up, corresponding to at least 5-fold reduction. Only 13.02% of jobs experience a slowdown, with the largest slowdown being 346%, corresponding to 4.46 times the completion time under HFS.

5) Sensitivity Analysis: Table IX shows the average map and job completion time of Pandas-accelerated Fair with threshold values 5, 8 and 10 when the average slowdown observed is 8. Even with an inaccurate threshold, we observe 2.8-fold improvement over HFS in average job completion time while we achieve 3.38-fold improvement with the correct threshold.

D. Scheduler Overhead

The space overhead of Pandas is negligible as our data structures maintain pointers to tasks, rather than keeping multiple copies. Table X shows the average scheduling delay for the trace with 192 jobs on the private cluster. We observe that the delay is comparable across the schedulers, with Pandas-accelerated FIFO being the fastest.

VIII. CONCLUSION

In this paper, we proposed Pandas, a robust locality-aware task-level accelerator with provable stochastic delay optimality. We demonstrated that Pandas significantly accelerates the Hadoop schedulers in the presence of hot-spots.

REFERENCES

[1] *Amazon EC2*. (Sep. 24, 2016). [Online]. Available: http://aws.amazon.com

[2] *OpenStack*. (Sep. 24, 2016). [Online]. Available: http://www.openstack.org

[3] C. L. Abad, Y. Lu, and R. H. Campbell, "DARE: Adaptive data replication for efficient cluster scheduling," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2011, pp. 159–168.

[4] C. Abad, N. Roberts, Y. Lu, and R. H. Campbell, "A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns," in *Proc. Int. Symp. Workload Characterization (IISWC)*, 2012, pp. 100–109.

[5] G. Ananthanarayanan *et al.*, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proc. Conf. Comput. Syst. (EuroSys)*, 2011, pp. 287–300.

[6] G. Ananthanarayanan *et al.*, "Reining in the outliers in MapReduce clusters using Mantri," in *Proc. Oper. Syst. Design Implement. (USENIX)*, 2010, pp. 1–14.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 185–198.

[8] G. Ananthanarayanan *et al.*, "PACMan: Coordinated memory caching for parallel jobs," in *Proc. Conf. Networked Syst. Design Implement.*, 2012, pp. 20–20.

[9] X. Bu, J. Rao, and C.-Z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 227–238.

[10] F. Chen, M. Kodialam, and T. V. Lakshman, "Joint scheduling of processing and shuffle phases in MapReduce systems.," in *Proc. IEEE INFOCOM*, Mar. 2012,pp. 1143–1151.

[11] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads," in *Proc. VLDB Endowment*, 2012, pp. 1802–1813.

[12] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. IEEE Annu. Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst. (MASCOTS)*, 2011, pp. 390–399.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Symp. Oper. Syst. Design Implement. (OSDI)*, 2004, pp. 1–13.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system.," in *Proc. ACM SIGOPS Oper. Syst. Rev.*, 2003. pp. 29–43.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 455–466.

[16] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-gravity reduce task scheduling to lower MapReduce network traffic," in *Proc. IEEE Cloud Comput. Int. Conf. (CLOUD)*, 2012, pp. 49–58.

[17] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique.," in *Proc. Cloud Comput. Technol. Sci. (CloudCom)*, 2011, pp. 40–47.

[18] S. Ibrahim *et al.*, "Maestro: Replica-aware map scheduling for mapreduce," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, 2012, pp. 435–442.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. Eur. Conf. Comput. Syst.*, Jun. 2007, pp. 59–72.

[20] M. Isard *et al.*, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proc. Symp. Operating Syst. Principles (ACM SOSP)*, 2009, pp. 261–276.

[21] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "Bar: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proc. Int. Symp. Cluster, Cloud Grid Computing (CCGrid)*, 2011, pp. 295–304.

[22] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in MapReduce," *Perform. Eval.*, vol. 70, no, 10, pp. 720—735, 2013.

[23] P. Nguyen, T. Simon, M. Halem, D. Chapman, and Q. Le, "A hybrid scheduling algorithm for data intensive workloads in a mapreduce environment," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput.*, Nov. 2012, pp. 161–167.

[24] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for MapReduce in a cloud," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011.

[25] J. Polo *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters.," in *Proc. Middleware Conf.*, 2011, pp. 187–207.

[26] Q. Pu *et al.*, "Low latency geo-distributed data analytics," in *Proc. Conf. Special Interest Group Data Commun.*, 2015, pp. 421–434.

[27] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proc. ACM SIGCOMM*, 2015, pp. 379–392. [Online]. Available: http://users.cms.caltech.edu/adamw/papers/hopper.pdf

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, 2010, pp. 1–10.

[29] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential MapReduce jobs," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1627–1635.

[30] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for MapReduce resource-aware scheduling.," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1618–1626.

[31] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. Conf. Operating Syst. Design Implement.*, 2014, pp. 301–316.

[32] A. Verma, L. Cherkasova, and R. H. Campbell, "Two sides of a coin: Optimizing the schedule of MapReduce jobs to minimize their makespan and improve cluster performance," in *Proc. IEEE Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, Aug. 2012, pp. 11–18.

[33] A. Vulimiri *et al.*, "Global analytics in the face of bandwidth and regulatory constraints," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, 2015, pp. 323–336.

[34] C. Wang *et al.*, "OPTAS: Optimal data placement in mapreduce," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2013, pp. 315–322.

[35] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 190–203, Feb. 2013.

[36] J. Wolf *et al.*, "Flex: A slot allocation scheduling optimizer for Map Reduce workloads.," in *Proc. Middleware Conf.*, 2010, pp. 1–20.

[37] Q. Xie and Y. Lu, "Priority algorithm for near-data scheduling: Throughput and heavy-traffic optimality," in *Proc.IEEE Conf. Comput. Commun. (INFOCOM)*, May 2015, pp. 963–972.

[38] Q. Xie and Y. Lu, "Priority algorithm for near-data scheduling: Throughput and heavy-traffic optimality," Tech Rep., 2015. [Online]. Available: http://simula.csl.illinois.edu/wp-content/uploads/2015/05/near-data-long.pdf

[39] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified relational data processing on large clusters," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2007, pp. 1029–1040.

[40] Y. Yao, J. Tai, B. Sheng, and N. Mi, "LsPS: A job size-based scheduler for efficient task assignments in hadoop," *IEEE Trans. Cloud Comput.*, vol. 3, no. 4, pp. 411–424, Oct./Dec. 2014.

[41] *Resource-Centric Compression in AM-RM Protocol Limits Scheduling*, accessed on Jul. 2, 2015. [Online]. Available: https://issues.apache.org/jira/browse/YARN-371

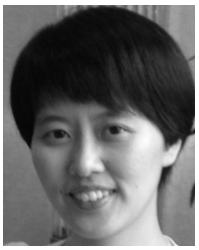
[42] *Support Delay Scheduling For Node Locality in MR2's Capacity Scheduler*, accessed on Jul. 2, 2015. [Online]. Available: https://issues.apache.org/jira/browse/YARN-80

[43] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2010, pp. 265–278.

[44] Y. Zhu *et al.*, "Minimizing makespan and total completion time in mapreduce-like systems," in *Proc. INFOCOM*, 2014, pp. 2166–2174.

Qiaomin Xie received the B.E. degree in electronic engineering from Tsinghua University in 2010, and the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, in 2016. Her research focuses on scalability, multiresource packing, and scheduling of efficient data-centric systems. She is the recipient of the Yi-Min Wang and Pi-Yu Chung Research Award (2015) and the best paper award from the IFIP Performance Conference (2011).

Mayank Pundir received the B.Tech. degree from the Indraprastha Institute of Information Technology, Delhi, India, in 2013, and the M.S. degree in computer science from the University of Illinois at Urbana–Champaign in 2015, where he worked on scheduling, elasticity, and fault tolerance in distributed systems. He is currently a Software Engineer with Facebook working on distributed systems and algorithms.



Yi Lu is an Assistant Professor with the Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign. Her work focuses on developing scalable architectures and algorithms for large networking systems, such as modern Web services with dynamic content, data-centric computing, and social networks. Her work spans fundamental analysis and algorithm implementation, and emphasizes the design of low-complexity, easy-to-implement algorithms. She is a recipient of the Sigmetrics Best Paper Award in 2008, the Performance Best Paper Award in 2011, the NSF Career Award in 2012, and the Sigmetrics Rising Star Award in 2016.

Cristina L. Abad received the M.S. and Ph.D. degrees from the Computer Science Department, University of Illinois at Urbana–Champaign. She is an Assistant Professor with Escuela Superior Politecnica del Litoral, Ecuador. Her main research interests lie in the area of distributed systems. She was a recipient of the Computer Science Excellence Fellowship and a Fulbright Scholarship at the University of Illinois at Urbana–Champaign. She received a Google Faculty Research Award in 2016 to work in Big Data storage workload modeling.

Roy H. Campbell (M'84–SM'04–F'05–LF'17) received the B.S. degree (Hons.) in mathematics, with a minor in physics, from the University of Sussex in 1969, and the M.S. and Ph.D. degrees in computer science from Newcastle University, Newcastle upon Tyne, U.K., in 1972 and 1976, respectively. He is the Sohaib and Sara Abbasi Professor with the Department of Computer Science, Director of Graduate Programs, Director of the NSA Designated Center for Academic Excellence and Research in Information Assurance, Director of the Air Force funded Assured Cloud Computing Center, Information Trust Institute, and the 2013-2014 Chair of the University Senate at the University of Illinois at Urbana-Champaign. His research interests are the problems, engineering, and construction techniques of complex system software: Cloud computing, big data, security, distributed systems, continuous media, and real-time control pose system challenges, especially to operating system designers.