2018 IEEE International Parallel and Distributed Processing Symposium

Characterizing Scheduling Delay for Low-latency Data Analytics Workloads

Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou
University of Colorado, Colorado Springs, USA
Emails: {cwei,epi,swang,xzhou}@uccs.edu

***Abstract*—Data analytics workloads are shifting to shorter task execution time, higher degree of parallelism, and execution on faster hardware. As a result, job scheduling is becoming a bottleneck, which needs to offer extreme low-latency, massive throughput, and high scalability. However, few efforts have been focused on systematically understanding the scheduling delay.**

In this paper, we propose a method and develop a tool, *SD-checker*, that decomposes the job scheduling delay into multiple components and characterizes each by extensive experiments. SDchecker extracts event messages through mining both cluster scheduler logs and application logs, and constructs a scheduling order graph for the ease of analysis. We use SDchecker to evaluate Spark-SQL on a popular cluster scheduler Yarn. Results show that the scheduling delay may account for 60% of the job runtime of small data analytics workloads. After decomposing the total scheduling delay, we find Spark itself contributes 70% of the delay. Through the evaluation and analysis, we conclude that (1) The causes of scheduling delay are determined by many factors, and (2) The job scheduling is not well optimized yet, and far from ideal for low-latency data analytics workloads.

I. Introduction

Spurred by demands for low-latency interactive data processing, there have been an increasing number of short data analytics jobs scheduled and managed by cluster schedulers. This trend is still continuing to enable more sophisticated parallel data processing. For example, in-memory computation application Spark [1], [2] has announced to target sub-second data queries in the future release.

To improve resource utilization, data analytics applications are often run in a multi-tenant cluster and managed by a cluster scheduler such as Apache Yarn [3] or Mesos [4]. The cluster scheduler handles the resource allocation, execution instances (e.g., Spark executor) placement and launching, while the application is responsible for scheduling tasks (e.g., Spark tasks) to execution instances. Low-latency data analytics applications are vulnerable to scheduling delay. First, when tasks finish within sub-seconds, resource allocation decisions must be made at high throughput. Second, task launching and localization delay may also become a bottleneck, as the application binary needs to be uploaded to the cluster and localized to the node on which is scheduled to run the task. Finally, for those cluster schedulers [3], [4] that apply a two-level scheduling policy, multiple levels of scheduling entities may further complicate the scheduling procedure of short jobs. Taking Spark-on-Yarn as an example, Spark's own task scheduling is unable to schedule tasks to individual executors, until the executors are allocated and launched on a node by YARN, which further prolongs the scheduling delay.

Many efforts in performance improvement of data analytics applications have been motivated by speeding up networks [5], [6], optimizing disks [7] and mitigating stragglers [8], [9]. There are also efforts on understanding the performance bottleneck in Spark [10], and building a prediction model for long running iterative jobs [11]. However, these efforts on addressing performance issues have a fundamental assumption that each job has a long running time, thus the scheduling delay can be neglected compared to the job execution time. We argue that this assumption will not hold true when a job is **tiny** and **short**.

There are studies [12], [13], [14], [15] that aim to address scheduling issues for short jobs as well as hybrid scheduling issues for both long and short jobs. However, these efforts only consider the scheduling decision latency, but are oblivious to the delay from other components of the system such as container launching and application initializing. In practice, the scheduling delay spans the entire period from the application submission to when its first task is scheduled to run.

However, understanding the scheduling delay for short jobs is technically nontrivial. First, many loosely coupled and distributed modules are involved to schedule a job (or task). The complex communication protocols and RPC calls make it difficult to reveal both the order and source of events that mark the scheduling delay. Second, the intrusive method [16] by manually inserting trace points in source code is also tedious, since it requires tremendous efforts to understand the source code and causes overheads.

In this work, we propose to utilize cluster scheduler logs and application logs to capture and characterize the scheduling delay across both the cluster scheduler and applications in a non-intrusive manner. We leverage the fact that the cluster scheduler maintains *state machines* in its data structures that represent the scheduling entity and output logs on state changes. For example, in Yarn's NodeManager, a message will be logged in class `ContainerImpl` to record the container state transformation from `LOCALIZING` to `SCHEDULED`, which implies that time elapsed between the two states be the container localization delay. Based on the insight, we develop a tool, *SDchecker*, to offline mine these critical log messages from both the scheduler and application logs. SDchecker builds connections between correlated states and orders them based on their timestamps. There are workflows across multiple logs, for example, container allocation messages are logged by `ResourceManager` while container launching messages are logged by `NodeManager`. We group these workflows based on their global IDs, such as application ID

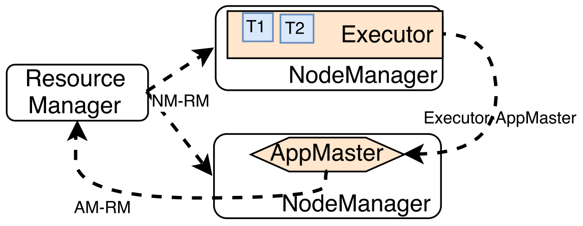

Fig. 1. The architecture of Spark-SQL on Apache Yarn. The AppMaster manages one executor. Two Spark tasks are scheduled in the executor. The dashed line represents the heartbeats between different components.

and container IDs. We finally construct a scheduling graph for each application, in which SDchecker decomposes scheduling delay into components for analysis.

We use SDchecker to evaluate Spark-SQL application [17], which aims to support sub-second queries, on cluster scheduler Apache Yarn. In this paper, we choose Spark-on-Yarn as our research platform, since these two frameworks are widely used in industry and are popular in open source community. We summarize our primary contributions as follows:

- We design and implement SDchecker, a log analysis tool that analyzes each component of the scheduling delay for data-analytics applications by building scheduling graphs from log files. It also finds a new bug in Spark.
- We conduct a comprehensive evaluation in a wide range of scenarios and show how the scheduling delay is affected by changing each factor. We present a quantitative analysis under each scenario and discuss its causes.
- We provide lessons learned and propose optimization suggestions for each type of the scheduling delay, which might be helpful to researchers in scheduler development and to users in application implementation.

Our tool SDchecker is opensource and available at: https://github.com/yncxcw/sdchecker.

II. Background

A. Two-level Scheduling

The two-level scheduling design has been implemented in both Yarn and Mesos. The fundamental idea is to split scheduling functionalities into *out-application* scheduling that is responsible for resource allocation and *in-application* scheduling that is responsible for task scheduling after resource is granted. This design simplifies the resource management in a multi-tenant cluster so that diverse applications can be managed by a set of uniform APIs.

Taking resource allocation and task scheduling of Spark-on-Yarn as an example in Figure 1, a user first submits an application request to ResourceManager, and ResourceManager initiates resource allocation upon this request through a user configured scheduler (e.g., Capacity Scheduler or Fair Scheduler). After resources are granted in terms of containers (ensembles of CPU and memory), ResourceManager will contact the chosen NodeManager to launch AppMaster. Note that AppMaster here is a driver process in Spark. Further, AppMaster is delegated to continue requesting resources for its executors from ResourceManager. Following the same procedure, ResourceManager allocates resources and piggybacks the granted containers to AppMaster. After receiving newly allocated containers, AppMaster launches executors by notifying NodeManagers. For executor launching, NodeManager first localizes the environmental resources that the executor relies on, including executable codes, dependent libraries and dependent files. It then configures necessary environmental variables and input/output directory, and executes the launching script to launch the task. So far, the *out-application* delay is the result of these intrinsic operations and protocols in Yarn.

After an executor has been launched and registered with AppMaster, AppMaster schedules tasks on this executor. However, Spark tasks cannot be scheduled as soon as the executor is launched. We find there is a significant delay between the executor launching and the first task allocation. We also notice a similar delay between the driver launching and the time when the driver registers with ResourceManager. We attribute these delays as *in-application* delay, as they are inside Spark. Figure 1 also shows heartbeats which are implemented by RPC between different components. For example, the NodeManager-ResourceManager heartbeat is used to inform ResourceManager the node liveness, containers status, and resource usage. AppMaster-ResourceManager heartbeat is used to inform ResourceManager the application progress and resource allocation/deallocation. AppMaster-task heartbeat helps AppMaster to track and schedule tasks in each executor.

B. Spark-SQL

Spark-SQL [17] is a library for Spark that enables users to execute queries with structured data. The core design of Spark is to keep intermediate results in memory for future reuse, which avoids redundant disk I/Os. Thus, its design goal is to support low-latency and high-scalability data analytics in clusters. The interplay between Spark's task scheduling and Yarn's container scheduling forms a good representative to study the intrinsic scheduling mechanism and delay.

III. Methodology

A. Identification of Log Message

Large-scale distributed systems use logs to record key messages that reflect internal state changes and events to help users diagnose bugs and performance issues. However, there are so abundant information generated from logs that data mining from the unstructured information is a tedious and painful process. There are recent efforts on efficiently mining logs [18], [19]. In this work, we only focus on scheduling related log messages in Yarn and Spark. Both Yarn and Spark use log4j library for logging. Each log message has the following format:

```
timestamp class log-message
```

Each `timestamp` has a precision of 1 millisecond, which is also the precision of SDchecker.

To ease the management of internal state and provide fast failover, Yarn uses state machine to model its scheduling entities. For example, each application has a

```
ResourceManager:
1.RMAppImpl: app-id State change from
             ACCEPTED to RUNNING on event = ATTEMPT_REGISTERED
2.RMAppAttemptImpl: app-id State change from
             LAUNCHED to RUNNING on event = REGISTERED
3.RMContainerImpl: cont-id Container Transitioned from
             NEW to ALLOCATED on event = START
NodeManager:
4.ContainerImpl: Container cont-id transitioned from
             NEW to LOCALIZING on event = INIT_CONTAINER
Spark-Driver:
5.YarnRMClient: Registering the ApplicationMaster
Spark-Executor:
6.CoarseGrainedExecutorBackend: Got assigned task id
```

Fig. 2. A snippet of log output from ResourceManager, NodeManager, Spark-Driver and Spark-Executor. Note we did not show timestamps in the sample.

`RMAppImpl` class to represent a user submitted application. A typical state flow for `RMAppImpl` looks like: `NEW_SAVING->SUBMITTED->ACCEPTED->RUNNING ->FINAL_SAVING->FINISHED`, of which the state transformation from `ACCEPTED` to `RUNNING` is produced by the event `ATTEMPT_REGISTERED`. This event indicates the AppMaster registers with ResourceManager by the first heartbeat. Thus, the time interval between the very first `SUBMITTED` and `RUNNING` can be used to characterize the delay from application submission to AppMaster launching. Furthermore, all these state transformations are automatically logged either in ResourceManager log or in NodeManager logs, which enables us to study any component of the scheduling latency by dedicated log messages mining.

We give a snippet of logs from both Yarn and Spark in Figure 2. We highlight the extracted *states* to build the state changes and the extracted *IDs* to identify each application (or task). Based on the raw logs, we list the identified log messages and its explanation in Table I. For instance, log message 4 indicates that a container is allocated by ResourceManager, and log message 5 indicates that a container is acquired by an application that issues the resource request. In the table, there are two classes recording container states. `RMContainerImpl` is in ResourceManager while the other `ContainerImpl` is in NodeManager. We use container allocation related states in `RMContaienrImpl` to calculate the container acquisition delay, and container launching related states in `ContainerImpl` to measure the container launching delay. For Spark logs, we use the message that logs the registration with ResourceManager in the driver, i.e., log message 10 and 13, and the first message that logs the assigned task id, i.e., log messages 14, to measure the in-application delay. We also manually add two log messages, i.e., log messages 11 and 12, into Spark source code to capture the container allocation delay.

B. SDchecker

SDchecker is an offline log mining tool that is designed to understand the scheduling delay for data analytics applications. Its key objective is to build a scheduling graph for each application to decompose the scheduling delay into multiple components. To use this tool, users first need to run a bunch of data analytics applications using Yarn as the cluster scheduler. After these applications complete, SDchecker is able to collect

TABLE I
IDENTIFIED LOG MESSAGES IN OUR ANALYSIS

Class	State	Explanation
1.RMAppImpl	SUBMITTED	App registers
2.RMAppImpl	ACCEPTED	App to be scheduled
3.RMAppImpl	APT_REGISTERED	Appmaster registers
4.RMContainerImpl	ALLOCATED	Container is allocated
5.RMContainerImpl	ACQUIRED	Container is acquired
6.ContainerImpl	LOCALIZING	Container starts localizing
7.ContainerImpl	SCHEDULED	Containder starts scheduling
8.ContainerImpl	RUNNING	Container starts running
9.Spark-Driver	FISRT_LOG	First log in Spark-driver
10.Spark-Driver	REGISTER	Spark-driver registers
11.Spark-Driver	START_ALLO	Start requesting containers
12.Spark-Driver	END_ALLO	Requested containders met
13.Spark-Executor	FIRST_LOG	First log in Spark-executor
14.Spark-Executor	FIRST_TASK	First task assigned

both Yarn's logs and applications' logs, and parse the logs to extract scheduling related messages using regular expression. Note that there are many state transformations for one scheduling entity. We only focus on those states that are critical for the scheduling delay analysis.

For each container, NodeManager spawns a thread to execute the launching script and this thread will run into a blocking call until the completion of the container. Thus, NodeManager cannot tell when the container is actually launched. As a result, we use the first log message (log messages 9 and 13) to mark the successful launching of the Spark driver and Spark executor. In addition, the resource requests may be queued and the scheduling delay could be significantly increased for an overloaded cluster, in which case ResourceManager will act as an arbiter to prioritize some resource requests and delay the others. We do not consider the queueing delay in this work, but focus on the scheduling delay caused by the system itself.

C. Scheduling Graph

As shown in Figure 2, SDchecker binds each log event with its corresponding global ID (application ID or container ID). SDchecker then aggregates and groups state transformations based on the IDs. After sorting collected states using their timestamps from the same container or application, SDchecker builds a scheduling graph as shown in Figure 3. The figure shows a scheduling graph for a Spark application with two executors and each executor has one task. We mark the states with the log message numbers from Table I. To assist analysis, we define several types of delays as follows:

- **Total scheduling delay:** It captures the delay between the time when the application is submitted to ResourceManager and when the first user-defined task is assigned. It includes Yarn caused and Spark caused delays.
- **AM delay:** It represents the delay caused by AppMaster scheduling and launching.
- **Cf delay and Cl delay:** Two metrics refer to the delay from application submission to the first container launching and to the last container launching, respectively.

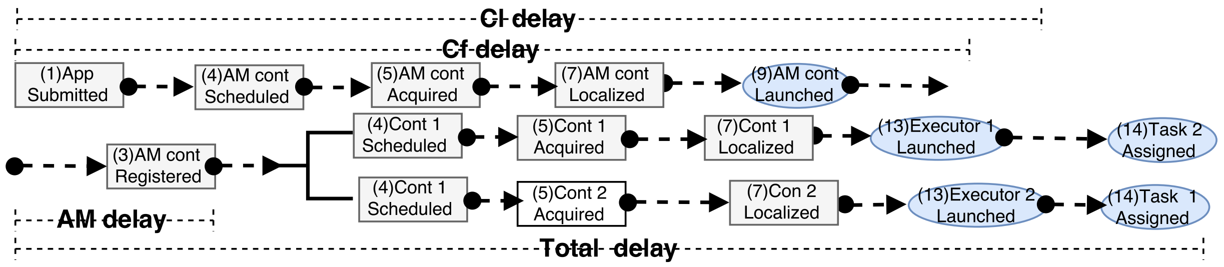

Fig. 3. The scheduling graph for a spark application with two executors and each executor has one task. The rectangles represent the delay caused by Yarn while circles represent the delay caused by Spark. The log message number from Table I is marked with each state.

- **Out-application delay and In-application delay:** Two metrics are used to inspect which component of delay is caused by Yarn and which component is caused by Spark.

We also study each component of the delay that is decoupled from the total scheduling delay. For example, the study of single container launching delay gives us a comprehensive understanding about how lengthy the executor launching is and its impact to the total scheduling delay.

IV. Evaluation and Characterization

A. Experimental Setup

We implemented SDChecker and conducted evaluations on a 26-node cluster. Each node has two 8-core Intel Xeon E5-2640 processors with hyper-threading enabled, 132GB of RAM, and 5x1-TB hard drivers configured as RAID-5. The machines are interconnected by 10Gbps Ethernet. Each node runs the Ubuntu-16.04. One node is configured as the NTP server for clock synchronization. To evaluate both centralized scheduler and decentralized scheduler, we deployed Hadoop-3.0.0-alpha3 in the cluster. The Hadoop-3.0 version implements the hybrid scheduler idea introduced in Mercury [14]. It supports both a centralized scheduler (e.g., Capacity Scheduler and Fair Scheduler) and a newly added distributed scheduler to satisfy the scheduling quality and scheduling latency demands. More specifically, the new distributed scheduler includes features of a decentralized opportunistic scheduler and opportunistic containers. Without losing generality, we use the Capacity Scheduler in the evaluation. We configure Hadoop distributed file system (HDFS) with a replication factor of three and a block size of 128MB. We use Spark-2.2.0 for evaluation of Spark-SQL. To make Spark aware of the opportunistic containers in decentralized scheduling, we made minor modifications to Spark source code [1]. As OS-based lightweight virtualization, e.g., Docker and LXC, is becoming popular, we also evaluate how the new technique impacts the scheduling delay. To evaluate the launching overhead with Docker containers, we use Docker-1.12.1 to create containers.

We use TPC-H on Spark-SQL as the low-latency data analytics workloads. Hive [20] is used to populate TPC-H tables in HDFS. We configure each Spark executor with 4GB memory and 8 cores. Unless otherwise stated, we configure each Spark-SQL with four executors and the input data size 2GB. However, we also vary the input size and the number of containers so as to study how application characteristics affect the scheduling delay.

We use two subsets of google-trace [21] to simulate the production query submission pattern. The long trace contains 2,000 queries, and is used to report the overall scheduling delays. The short trace contains 200 queries, and is used to study each component of the overall scheduling delays.

B. Overall Scheduling Delays

We first give the overall scheduling delays for the 2000 TPC-H queries with 2GB input data. We configured each Spark-SQL with four executors. Figure 4 shows the job runtime (*job*), total scheduling delay (*total*), AM delay (*am*), in-application delay (*in*) and out-application delay (*out*).

Observations Figure 4-(a) shows that the 95_{th} percentile latency for *total*, *am*, *in* and *out* are 17.2s, 6s, 12.7s and 5.3s, respectively. The job runtime varies across different queries. Figure 4-(b) shows the normalized delays, in which the *total* is normalized to the job runtime, *am*, *in* and *out* are normalized to *total*. We find about **40%** of the job runtime is spent on the scheduling delay, and it approaches 60% in the worst case. This means that the scheduling delay causes significant overhead for low-latency data analytics workloads and has became a bottleneck for short jobs. Within the total scheduling delay, more than 70% is attributed to *in* (Spark) and only less than 30% is attributed to *out* (Yarn). This finding reveals the fact that scheduling in Spark is still not well optimized for drivers, executor initializing, and task scheduling. Note that we omit the Spark scheduling delay during job execution for the tasks in subsequent stages, because this delay component would be overlapped with the task execution.

As shown in Figure 4-(c), the overall scheduling delays have significant variances. We believe the variances are the result of interplay of many loosely coupled components which direct the resource allocation and task scheduling. Thus, one delayed component will be amplified in other components that affects the overall scheduling performance. We notice that *in* varies more widely than *out* and contributes most to the variance of *total*. We examine the cause in subsequent section IV-E.

The AM delay *am* contributes around 35% of the total scheduling delay. This component of the delay is the result of both Spark driver launching and Yarn scheduling. Furthermore, AppMaster could be stuck in resource allocation when the cluster is almost fully utilized [22], in which case AppMaster's

[1]Source code is available at: https://github.com/yncxcw/spark-2.0

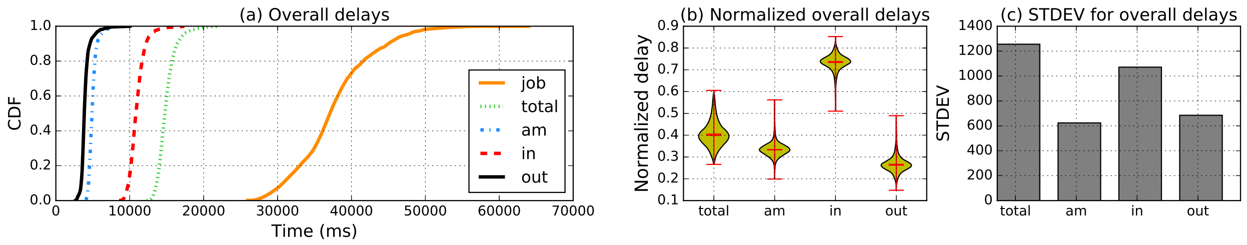

Fig. 4. Overall scheduling delays for 2000 Spark-SQL query jobs on a 2GB dataset. Figure shows the job runtime (*job*), total scheduling delay (*total*), AM delay (*am*), in-application delay (*in*), and out-application delay (*out*). (a) shows the CDF of the delays. (b) shows the normalized delays. The total scheduling delay is normalized to the job runtime; in-application delay, out-application delay and AM delay are normalized to the total scheduling delay. (c) shows the standard deviation of the delays.

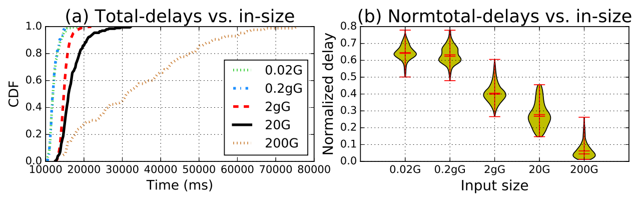

Fig. 5. (a) shows the total scheduling delays with various input data sizes. (b) shows the normalized total scheduling delays (to job runtime) with various input data sizes.

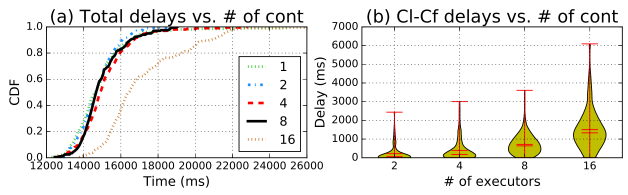

Fig. 6. (a) shows the total scheduling delays with various numbers of executors. (b) shows the $CI - Cf$ delay between the first launched container time and the last launched container time, with various number of executors.

container request is queued and allocated by ResourceManager to satisfy cluster-wide fairness.

Impact of the job size on the scheduling delay We study how the job size affects the scheduling delay and show the results in Figure 5. We increase the job size by giving different size of TPC-H dataset, ranging from 20MB to 200GB. We obtained two observations in this experiment. First, the normalized total scheduling delay decreases with the increase of job size. That' is due to the fact that a larger job size leads to a longer job runtime. However, for an extremely tiny job size (e.g., 20MB), the scheduling delay accounts for more than 65% of the job runtime, and almost 80% in the worst case. Again, we demonstrate that existing cluster schedulers are too expensive for small data analytics workloads.

Second, we observe a trend that larger input size leads to a longer total scheduling delay. For example, Figure 5 shows the 95_{th} percentile of the total scheduling delay for 200GB input size is 60.4s, 4x longer than that of 20MB input size. We also notice that the CDF for 200GB input size presents a heavy-tail distribution. We further examine other metrics and find that with 200GB data input the *out* delay is deteriorated by 1.5x while *in* delay is deteriorated by 5.7x than the respective delays with 20MB data input.

In our experiment setting, Yarn localizes application binary (java jar) at /yarn-temp directory and the directory lies in the same hard drive that HDFS resides. We infer the deteriorated total scheduling delay is caused by intensive cluster-wide I/O interference where executor launching is competing with task data reading from HDFS. We further examine how I/O interference affects scheduling in section IV-E.

Impact of number of executors on the scheduling delay In parallel data processing, generally higher parallelism brings larger throughput and better performance. Thus, to achieve larger task fanout and to reduce the number of waves of iterations, Spark users may configure a large number of executors. However, there exists a trade-off between the scheduling delay and parallelism.

As shown in Figure 6, Spark-SQL with 16 executors experiences a 21.5s of 95_{th} percentile tail latency which is 4s longer than that with 8 executors. Its also show the $Cl - Cf$ delay which captures the duration between first container launching and last container launching. Obviously, a larger number of executors leads to both longer delay and higher variance.

In default Yarn mode, Spark does not initiate task scheduling until a certain number of executors are granted by Yarn (In Spark, 80% of requested executors). Thus, the more executors the Spark requests, the longer waiting time the spark driver is charged before Spark initiates task scheduling. Further, executor requesting and allocation involve many modules. Requesting more executors adds more performance variance. This waiting period contributes to the total scheduling delay.

The situations get even worse in a highly loaded cluster. Consider that a Spark application requests N containers but only gets M ($M < N$). The Spark driver cannot move forward to task execution until other applications finish and their occupied resources are released. In this case, the resource waiting time is highly unpredictable depending on the system load. We omit this scenario since we only consider moderate cluster loads and focus on cluster scheduling delay itself.

C. Out-Application Delay

In this section, we study each component of the scheduling delay caused by Yarn, i.e., resource allocation delay, container acquisition delay, localization delay, and launching delay.

Impact of schedulers on the resource allocation delay Compared to a centralized scheduler, recently introduced decentral-

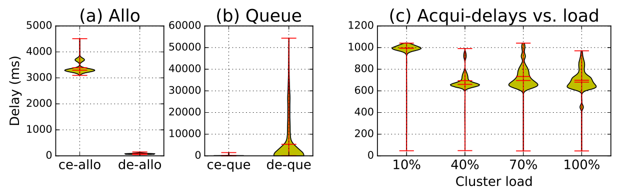

Fig. 7. (a)-(b) shows the accumulated container allocation delay and queuing delay by a centralized scheduler (ce-) and a distributed scheduler (de-). (c) shows the container acquisition delay with various cluster loads.

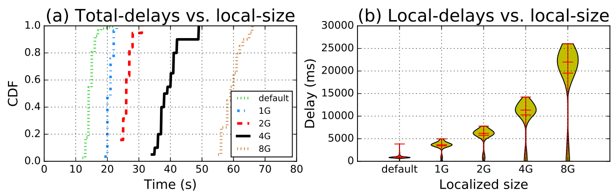

Fig. 8. (a) shows the total scheduling delays with various localized file sizes. (b) shows the localization delays with various localized file sizes.

TABLE II
CLUSTER CONTAINER THROUGHPUT UNDER VARIOUS WORKLOADS.

Cluster load	10%	40%	70%	100%
Throughput (1/s)	272	1056	1607	2831

ized schedulers [13], [12], [14] generally have the advantage of higher scalability and lower resource allocation delay. Since Yarn containers are requested and allocated in a batch mode, in this experiment, we adopt the Capacity Scheduler (centralized) and Opportunistic Scheduler (distributed), and use log messages 11 and 12 to calculate the aggregated resource allocation delay, that is, the delay AppMaster accumulates until all requested containers are granted.

As shown in Figure 7-(a), the distributed scheduler is almost 80x faster than the centralized scheduler in the median latency. Further, the 95_{th} percentile latency by the distributed scheduler and the centralized scheduler are 108ms and 3,709ms, respectively. However, as discussed by prior work [12], [14], while a distributed scheduler may achieve lower latency, it could also make poor scheduling decisions that thrash the job performance. Since a distributed scheduler uses a random algorithm to choose a slave node for each task, it lacks the global cluster information. Tasks scheduled by a distributed scheduler will be queued at the slave node, if the randomly picked node does not have enough resource to run the task. We demonstrate the scenario in Figure 7-(b). We create a highly loaded cluster and measure the task queuing delay (the interval after task localization and before task running). The figure shows tasks suffers from as much 53s queueing delay by a distributed scheduler, which almost equals to the average job completion time of TPC-H. The queuing delay for a centralized scheduler is just about 100ms. In the following experiments, we use the Capacity Scheduler.

Impact of cluster load on the container acquisition delay Previous research found the resource allocation delay of short jobs is significantly increased when the cluster is under high load [12]. In this experiment, we investigate how cluster load affects the resource allocation delay. We use MapReduce *wordcount* as the benchmark with different input sizes, since MapReduce will spawn a large number of map tasks that can quickly occupy the cluster resource. By scaling the input data size, we control the cluster load.

Table II shows the container allocation throughput. The Capacity Scheduler achieves a throughput of 2,831 containers per second when the cluster is fully utilized. We observe that the container allocation throughput scales well, which implies that the resource allocation delay does not increase with the cluster load. The Capacity Scheduler is still scalable even when the cluster is almost under 100% utilization. We believe it is due to the limitation of the cluster size (25 working nodes).

Next, we study a new type of delay, *container acquisition* delay. The container acquisition delay, calculated by using log messages 4 and 5, is used to capture the period that the container is allocated but not yet acquired by AppMaster. AppMaster communicates with ResourceManager to allocate and acquire containers through heartbeats.

Figure 7-(c) shows one interesting finding, that is, the container acquisition delays are capped by one second, which is the default heartbeat interval for MapReduce. We also notice that the container acquisition delays have very high variances, because it is highly correlated to the heartbeat interval. Thus, we can draw a trade-off here, that is, higher frequency of heartbeats lowers the container acquisition delay, but at the risk of overwhelming the cluster network.

Impact of localized file size on the localization delay Cluster schedulers are required to distribute user submitted *localization files*, typically binary packages or dependent files, to the target nodes where the tasks are scheduled. The localization refers to the process that the container downloads localization files from other storage medium (e.g., shared file systems or shared caches) before the job binary is launched. Spark uses HDFS to cache the localization files. These files should be localized to the local file system before executors are launched.

We evaluate how the localized file size affects the scheduling delay. We use Spark submit command "-f" option to force the application to upload additional files with various sizes. SDchecker uses log messages 6 and 7 to calculate the localization delay. As shown in Figure 8, the localization delay increases quickly with the increase of localized file size. In the experimental setting, the default Spark-SQL takes around 500ms to localize a 500MB file package, including Spark jar, TPC-H jar, and Spark configuration files. However, for a 8GB localized file, the localization delay is about 23s. As a result, the total scheduling delay is severely deteriorated for large localized files. Note, we also observed that some localization delays are less than one second with 8GB localized file size, which is due to the driver localization.

Impact of instance type on the launching delay Launching delay describes the period from that the NodeManager invokes the launching script to that the launched instance logs the first message. SDchecker uses log messages 7 and 8 to record this delay. In this experiment, we run different applications and record their launching delays based on the instance type of

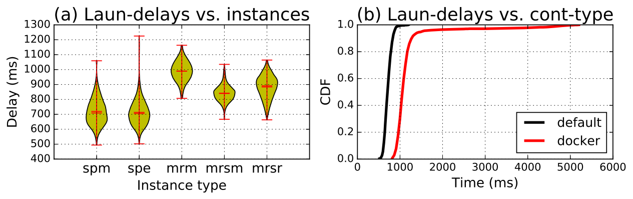

Fig. 9. (a) shows the launching delays with different instance types. (b) shows the launching delays with the default Yarn container and Docker container.

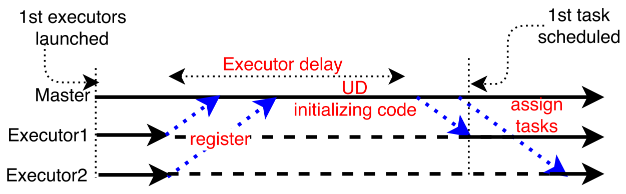

Fig. 10. The workflow to demonstrate the Spark task scheduling and to understand the executor delays. The solid lines denote running state, while the dashed lines denote the idleness.

each container. The instance types are Spark driver (spm), Spark executor (spe), MapReduce master (mrm), MapReduce map task (mrsm) and MapReduce reduce task (mrsr).

Figure 9-(a) shows that the median latency for Spark executor and Spark driver is about 700ms. We also notice that MapReduce causes a bit longer launching latency than Spark does. Thus, different instance types have various impacts on the launching delay.

Impact of container type on the launching delay Yarn uses container as an abstraction of resource. Recently proposed Docker container technique further strengthens this abstraction by leveraging lightweight virtualization to enforce strict resource isolation and security. Since Docker and Linux containers are getting more popular, we expect more applications and services will be deployed with this new technique. Therefore, the overhead caused by Docker is critical for low-latency data analytic workloads. We evaluate the launching delay for Spark-SQL with and without Docker. Figure 9-(b) shows that the Docker container causes a launching overhead of 350ms for median latency and 658ms for the 95_{th} percentile latency. As discussed in [23], the launching overhead of Docker is caused by loading the image from the local hub and mounting it to a predefined path. The image we used here has a size of 2.65GB. We anticipate that the overhead will be larger if more files and packages are built into the image. Another observation is that application launching with Docker presents a long-tail effect. We think the source of the variance is from the extra I/O operations when launching applications with Docker.

D. In-Application Delay

The in-application delay is used to attribute the time Spark spends on user application initialization and task scheduling. It is composed of *driver delay* that is caused by driver initialization, and *executor delay* that is caused by further application initialization and task scheduling. For the driver delay, it can be interpreted as the time that the driver spends on

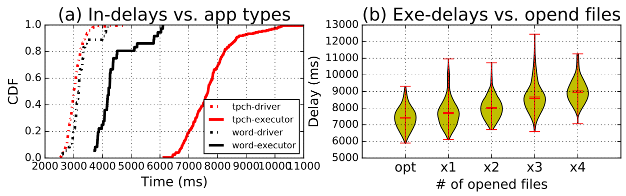

Fig. 11. (a) shows the in-application delay for Spark wordcount and Spark-SQL. (b) shows the executor delay for optimized Spark-SQL, default Spark-SQL and Spark-SQL with various number of opened files. In X-axis, opt is the optimized version, x1 is the default case, x2 means doubling of the number of opened files, and so on.

initialization after the driver is launched by NodeManager but before the driver registers with ResourceManager. However, the executor delay is not so intuitive. As shown in Figure 10, the executor delay covers the period from the moment when the first executor is launched to the moment when the first task is scheduled. This period spans the process of executor registration with the driver, user application initialization, and task scheduling. The executors will remain idle during the period of user initialization at the driver. The reason is that the executors are waiting for the tasks to be assigned from the driver. Note that the time before containers are launched and after the driver registers with ResourceManager is categorized into the out-application delay.

We use log messages 9 and 10 to calculate the driver delay, and log messages 13 and 14 to calculate the executor delay. Figure 11-(a) shows the driver delay and executor delay for Spark wordcount and Spark-SQL, respectively. The driver delays for both wordcount and Spark-SQL are almost identical, both experiencing a latency around 3s. However, Spark-SQL suffers from a longer executor delay. The 95_{th} percentile executor delay is 6.0s for wordcount and 9.5s for Spark-SQL. As the fact, all Spark applications share the same code for driver initialization and container requesting, but significantly differ in user defined initialization. We believe that the performance discrepancy is caused by user defined code during application initialization.

Code optimization We further compared the source code of Spark-SQL and Spark wordcount, and found there are more new initialized Resilient Distributed Datasets (RDDs [2]) in Spark-SQL than in Spark wordcount. There are eight tables in TPC-H, and each table will be initialized from a file and transformed to a RDD. But there is only one file needs to be loaded for Spark wordcount. Furthermore, for each newly defined RDD that is associated with a file, Spark will use a broadcast variable to store it. The cost of creating a broadcast variable is expensive, and more significantly this cost lies in the critical path of scheduling. To verify the analysis, we modified the source code of TPC-H on Spark to evaluate the executor delay under different number of opened files. As shown in Figure 11-(b), with more opened files during user application initialization, the executor delay gets longer. We optimized the RDD initialization code by using Scala *Future* objects so as to parallel the operations. Results show that the optimized version of TPC-H achieves a 2s reduction in the tail executor delay compared to the default.

This experiment provides the insight that the user initialization code lies in the critical path of job scheduling, so that application developers should try to avoid expensive operations or try to parallel them. Despite of the optimization, the in-application scheduling cost is still expensive.

E. *Performance Interference*

Impact of IO interference on the scheduling delay We first study how the scheduling delay is affected by IO interference. We use dfsIO that is provided in Hadoop test package to generate HDFS write interference that overloads both disks and the network in the cluster. The dfsIO spawns parallel map tasks to write data into HDFS. Each map task writes 20GB data. We thus control the number of parallel map tasks to control the intensity of IO interference.

Figure 12 shows the experimental results. Figure 12-(a) shows the comparison between the default case and the case with 100 map tasks as IO interference (100-interference). The 95_{th} percentile of the total scheduling delay for Spark-SQL is degraded by a factor of 3.9x. We further break down the result, and find both the in-application delay and the out-application delay experienced performance penalty. We study each component of the delay as we described previously. The results suggest the localization delay and the executor delay contribute most of the total scheduling delay.

The localization delay is a component of the out-application delay. As shown in Figure 12-(b), the localization delay increases as the IO interference degree increases. For 100-interference, the tail latency is 35s (7x slowdown), and the median latency is 4.7s (9.4x slowdown). Since the localization requires both the driver and the executors to download jars from HDFS, the dfsIO workloads compete for the shared network and disk resources with localization threads. Moreover, the heavy-tail curve of the total scheduling delay in Figure 12-(a) resembles the curve in Figure 5-(a), which further verifies that the huge input size for Spark-SQL can also cause significant IO interference.

The executor delay is a component of the in-application delay. As shown in Figure 12-(c), both the tail latency and the median latency for the executor delay incur 2.5x to 3.5x performance degradation under IO interference. We also notice that the distribution of the executor delay under strong IO interference (100-interference) is much more scattered than that of the default case. It suggests the influence of IO interference on the executor delay is more complicated and attributed to many factors. We provide two possible factors here: (1) heartbeats that executors used to register with the driver and assign Spark tasks can be blocked under network interference; (2) heavy disk activities interfere with JVM warm-up when the JVM is loading classes from jar packages.

One thing we should notice is that each Spark application has two interdependent localizations, the first one is to localize the driver jar and the second is to localize the executor jar. Therefore, the total scheduling delay is affected twice. As shown in Figure 12-(d), the AM delay is also exacerbated by as much as 8x, due to the result of driver localization.

Impact of CPU interference on the scheduling delay We then examine how CPU interference affects the scheduling delay. To generate the CPU interference, we use Kmeans as the benchmark from HiBench [24]. Kmeans is an iterative machine learning application, which always traverses on the same data set during iterations. To fully overload node's CPU resource, we configure each Kmeans application with 4 executors and each executor with 16 vcores. By managing the number of parallel Kmeans applications, we generate different levels of CPU interference.

As shown in Figure 13-(a), the 95_{th} percentile of the total scheduling delay suffers as much as 1.6x performance degradation when running 16 Kmeans applications as CPU interference (16-Kmeans app). However, unlike IO interference, only the in-application delay is severely affected under CPU interference. By Figures 13-(b) and 13-(c), we note both driver delay and executor delay experienced performance degradation, suffering as much as 2.9x and 2.4x slowdown with 16-Kmeans applications, respectively. These results are reasonable because: (1) both executor launching and driver launching are CPU intensive; (2) the CPU interference slows down the JVM warm-up when JVM interprets bytecode, executing JIT-compiled methods. We also observe that CPU interference further slows down the entire Spark-SQL execution, because most data analytics applications are CPU intensive.

Figure 13-(d) shows that the localization delay is moderately affected. Specifically the median latency suffers 1.4x slowdown. The reason is that the localization requires HDFS client to contact with a name node to query block information, and this process is CPU intensive. As the entire localization is dominated by IO, only slightly performance loss for the localization delay can be observed under strong CPU interference.

The two experiments also demonstrate that the in-application delay is more vulnerable to CPU interference than the out-application delay is, which explains the result that the in-application delay has more variations.

V. Discussion and Optimization

A. *Identification of a Bug*

We found a new bug by SDchecker during the evaluation process. When evaluating the resource allocation delay on the distributed scheduler using opportunistic containers, we found there are containers that were allocated but never used. This is identified due to the fact that many containers only log states related to NodeManager and ResourceManager but miss states logged by executor, e.g., log messages 13 and 14. This makes us to believe that Spark has requested more containers than its actual demand, which can be viewed as a bug. We reported this bug [25] to the Spark community and soon they confirmed our findings (duplicated by other finders).

B. *Proposed Optimizations*

Our analysis shows that the cluster scheduling is performed by many loosely coupled and distributed modules and, the scheduling delay is affected by many factors. Thus, it is almost impossible to have an unified solution. Instead, we propose a set of optimizations targeting at each component of the delay in Table III. For each delay, we list its source, cause, contribution to the total scheduling delay (results from section IV-B)

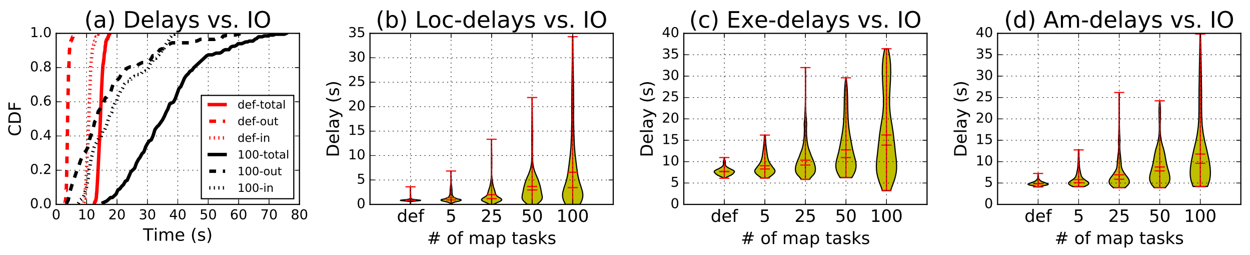

Fig. 12. (a) shows the overall delays with strong IO interference (100 map tasks). It shows two sets of results, the default case and the interference case, in total, out-application, and in-application delays. (b)-(d) show the localization delay, executor delay, and AM delay with various degrees of IO interference.

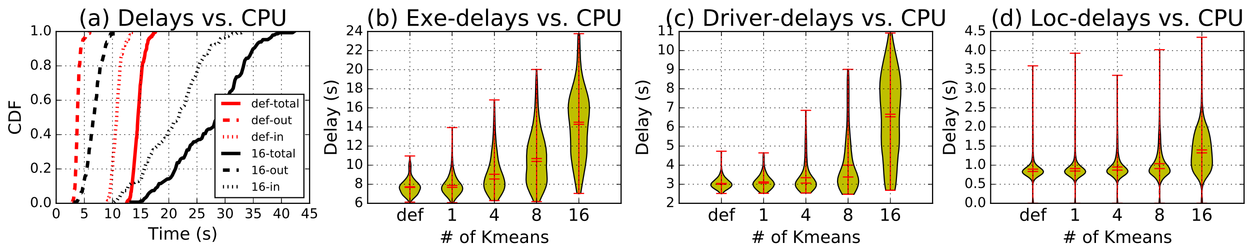

Fig. 13. (a) shows the overall delays with strong CPU interference (16 Kmeans). It shows two sets of results, the default case and the interference case, in total, out-application, and in-application delays. (b)-(d) show the executor delay, driver delay, and localization delay with various degrees of CPU interference.

TABLE III
SUMMARY OF THE SCHEDULING DELAYS AND PROPOSED FUTURE OPTIMIZATIONS

Source	Cause	%	Optmization
1.alloc-delays	Time of resource allocation decisions at ResourceManager	25%	Trade-off, using distributed scheduler
2.acqui-delays	Time of waiting allocated containers to be acquired by AppMaster	$< 1\%$	Trade-off, increasing heartbeat frequency
3.local-delays	Time of downloading localization files from HDFS	$< 1\%$	User&Design, dedicated storage&caching service
4.laun-delays	Time of launching AppMaster/executor (e.g., JVM starts)	$< 1\%$	User, avoiding OS-container
5.driver-delay	Time of Spark driver initialization	32%	Trade-off, JVM reuse
6.executor-delay	Time of Spark executor initialization and Spark task scheduling	41%	Trade-off&User, JVM reuse&user application optimizations

and proposed optimizations. We categorize the proposed future optimizations into three groups: trade-off (may cause some side-effects), user (optimizations is implemented on user-side, through application source code or configurations) and system (new design and implementation are required).

To address the resource allocation delay, a distributed scheduler is a tempting solution and has been used in production environments. However, a distributed scheduler may make poor decision because it lacks the global cluster state. Increasing the heartbeat frequency between AppMaster and ResourceManager alleviate the container acquisition delay but at the risk of overwhelming the entire cluster network. We have shown that the IO interference significantly thrashes the localization delay. One possible solution to address the IO interference is to isolate the localization IO from regular HDFS IO. For example, by setting up a HDFS storage class [26], users can configure the HDD storage class for regular IO and configure SDD or RAM disk storage class for localization IO. However, this approach only isolates the disk IO. The localization delay can still be bottlenecked by the network interference for those whose localization files locate on the remote nodes. Thus, we further propose to design a new caching service on each slave nodes. As a result, the recent most used localization files will be cached on local nodes in dedicated storage class, eliminating the effects of network interference. We leave this as our future work. Though the OS-container provides various benefits, including better resource isolation, ease of independence management and fast deployment, it also introduces the container launching overhead.

Driver delay and executor delay are caused during Spark initialization. Our analysis combined with previous literatures [27] make us realize these part of delays can be attributed to (1) complex and expensive operations at initialization and (2) JVM warm-up overhead. Users should try to avoid implementing expensive operations or to simplify the logics in application code to speed up the initializations. The JVM warm-up overhead refers to the time that JVM spends on JIT compilation and class loading at JVM starting phase. Based on [27], the JVM warm-up overhead is significant for short jobs, accounts for 30% of the job runtime for Spark-SQL. It also explains in-application delay is still significant though we optimize some expensive operations. One promising solution to address the JVM warm-up is to apply JVM reuse, which requires recurring applications.

VI. Related Work

Cluster Scheduling is a core component in data-intensive cloud computing. YARN [3] and Mesos [4] are two widely used open-source cluster managers. Both use a two-level architecture, decoupling allocation from application-specific logics such as task scheduling, speculative execution or failure handling. Another thread of work focuses on distributed schedulers to overcome the scalability problem in large-scale clusters. Sparrow [13] is a fully distributed scheduler that performs scheduling by performing randomized sampling. Hawk [12] and Mercury [14] both implement a hybrid scheduler to avoid inferior scheduling decisions for a subset of jobs as a trade-off between the scheduling quality and scalability. In this paper, we conducted the first comprehensive evaluation to study the scheduling delay for Spark-SQL on Yarn. Our results are also valuable for other platforms other than Yarn, since these cluster schedulers share similar working protocols.

Data analytics Applications Many data analytics frameworks have been developed. Among them, Hadoop [28] is developed to process massive data on commodity PCs. Spark [29] leverages the large memory to speed up data processing. Hive [20] provides SQL interface to access data based on Hadoop and Spark. Tez [30] provides users with strong expressions for DAG of tasks for data processing. In this paper, we focus on the data analytics workloads that require short latency. With the evolution of hardware and software, we anticipate more applications will be "shorter".

Characterize and Optimize Applications Many optimizations have been proposed and applied to data analytics applications. Delay scheduler [5] delays the scheduling of map tasks for better locality. Zaharia et al. conducted the first work to address stragglers in heterogeneous environments [8]. Ousterhout et al. provided a comprehensive study on performance issues on Spark applications [10]. However, none of the studies consider scheduling as a bottleneck.

VII. Conclusion

Understanding and characterizing the scheduling delay for low-latency data analytics applications is challenging but significant. In this paper, we proposed to use cluster schedule logs and applications logs to capture the job scheduling delay. We developed SDchekcer, a log mining tool that utilizes the state transformation log messages from Yarn and a few auxiliary logs from Spark to decompose the scheduling delay into components, and further analyzed each component and its cause by extensive evaluations. Our tool and experimental results reveal many important facts and insights, which can guide future scheduler design and performance optimization of low-latency data analytics workloads.

VIII. Acknowledgment

This research was supported in part by U.S. NSF award CNS-1422119.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." in *Proc. of USENIX HOTCLOUD*, 2010.
[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012.
[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. of ACM SoCC*, 2013.
[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proc. of USENIX NSDI*, 2011.
[5] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of ACM Eurosys*, 2010.
[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. of ACM SIGCOMM*, 2011.
[7] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. of ACM SoCC*, 2014.
[8] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments." in *Proc. of USENIX OSDI*, 2008.
[9] W. Chen, J. Rao, and X. Zhou, "Addressing performance heterogeneity in mapreduce clusters with elastic tasks," in *Proc. of IEEE IPDPS*, 2017.
[10] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks." in *Proc. of USENIX NSDI*, 2015.
[11] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *Proc. of USENIX NSDI*, 2016.
[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *Proc. of USENIX ATC*, 2015.
[13] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proc. of ACM SOSP*, 2013.
[14] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *Proc. of USENIX ATC*, 2015.
[15] W. Chen, J. Rao, and X. Zhou, "Preemptive, low latency datacenter scheduling via lightweight virtualization," in *Proc. of USENIX ATC*, 2017.
[16] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proc. of ACM SOSP*, 2015.
[17] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proc. of ACM SIGMOD*, 2015.
[18] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *Proc. of USENIX OSDI*, 2014.
[19] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle." in *Proc. of USENIX OSDI*, 2016.
[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. of VLDB Endowment*, 2009.
[21] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. of ACM SoCC*, 2012.
[22] D. Cheng, X. Zhou, P. Lama, J. Wu, and C. Jiang, "Cross-platform resource scheduling for spark and mapreduce on yarn," *IEEE Transactions on Computers*, vol. 66, no. 8, 2017.
[23] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers." in *Proc. of USENIX FAST*, 2016.
[24] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. of IEEE Data Engineering Workshops (ICDEW)*, 2010.
[25] "Spark-21562," https://issues.apache.org/jira/browse/SPARK-21562, 2017.
[26] "Archival storage, ssd and memory," https://hadoop.apache.org/docs/r2.8.0/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html, 2017.
[27] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems." in *Proc. of USENIX OSDI*, 2016.
[28] "Hadoop," *http://hadoop. apache. org*, 2009.
[29] A. Spark, "Lightning-fast cluster computing," 2013.
[30] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. of ACM SIGMOD*, 2015.