Eirene: Improving Short Job Latency Performance with Coordinated Cold Data Migration and Scheduler-Aware Task Cloning

Wei Zhou
University of Virginia
Charlottesville, USA
wz5ad@virginia.edu

K. Preston White
University of Virginia
Charlottesville, USA
kpwhite@virginia.edu

Hongfeng Yu
University of Nebraska-Lincoln
Lincoln, USA
yu@cse.unl.edu

***Abstract*—In large-scale enterprise data centers for big data analytics, long batched jobs and short interactive jobs are usually mixed. Hybrid job schedulers, consisting of one centralized scheduler for long jobs and multiple distributed schedulers for short jobs, have become a promising alternative because they can significantly shorten latencies of short jobs via independent and parallelized assignment of short tasks by distributed schedulers and lower chances of head-of-line blocking via a number of performance optimization techniques.**

However, short jobs are still facing long job latencies under hybrid job schedulers due to workload fluctuation and straggler task problem. In this paper, we propose *Eirene* to optimize the latency performance of short jobs via two schemes tightly coupled into the general architecture of hybrid job schedulers. *Coordinated Cold Data Migration* leverages high task waiting time of short jobs under heavily-loaded periods and migrates cold data from disks to local memory for the initial phase of reading input so as to shorten task runtime and queueing time. On the other hand, *Scheduler-Aware Task Cloning* exploits spare computing resources under lightly-loaded periods and performs proactive task cloning for short jobs to mitigate the straggler problem.

We implement a prototype of Eirene based on Eagle, a state-of-the-art hybrid job scheduler. Experimental results show that, under heavy loads, Eirene is able to improve 50-percentile (P50), 75-percentile (P75), 90-percentile (P90) latency performance of short jobs by up to 44.4%, 80.3%, 84.1% respectively compared with Eagle under the Facebook trace with a cluster of 50000 nodes.

***Index Terms*—Big Data, Job Scheduler, Resource Management**

I. Introduction

One salient characteristic of production workloads of big data analytics in large-scale data centers is a mixture of long jobs and short jobs, as evident in recent production workload analysis [1]–[5]. Although the total number of short jobs could be $10\times$ greater than that of long jobs, they usually consume disproportionally fewer resources than long jobs. For example, over 90% of jobs in Google clusters are short jobs, but short jobs consume only 17% of resources [3]. This is because computer clusters in an enterprise are usually shared by different departments for high utilization efficiency. It is common to see analysts and developers submit short but interactive jobs like ad-hoc queries or personalized search, while long-running services and batch jobs occupy a large portion of computing resources in shared clusters. Conventional centralized job schedulers like YARN [6] and Mesos [7] are able to achieve high cluster utilization since they have a global view of resource usage and resource demands of jobs. However, they are incapable of meeting the job latency[1] demand for short jobs because job queueing delay becomes non-trivial when the single scheduler is overwhelmed by a massive amount of short jobs [8]. In contrast, distributed job schedulers like Sparrow [8] are able to achieve good latency performance of short jobs with parallel job scheduling. However, they suffer from inefficient cluster utilization because decisions of task assignments are made based on sampling and randomization.

To this end, hybrid job schedulers like Hawk [9], Eagle [10], Phoenix [11], and Dice [12] have been proposed to deliver both high cluster utilization and low latency for short jobs with a combination of one centralized scheduler for long jobs and multiple distributed schedulers for short jobs, and become a promising alternative to conventional centralized schedulers and distributed schedulers. However, from existing workload analysis and our experimental study, we believe that short jobs under hybrid job schedulers still encounter severe performance issues due to two reasons. First, the widely-observed straggler problem can significantly degrade latency performance of short jobs as straggler tasks could take up to 8 times longer than the mean task runtime in Hadoop clusters, causing the jobs to be slowed down by 47% on average [13] (See Section II-D for more detail). Second, our experimental study shows that, even with hybrid job schedulers, workload fluctuation can still result in up to 3000 seconds of task waiting time for short jobs under heavily-loaded periods with the Yahoo trace, considering the mean task runtime of short jobs is less than 90.58 seconds (See Section II-B for more detail).

In this paper, we propose Eirene to address the above performance issues of short jobs with two schemes. On one

[1]Job latency, also called job completion delay, which denotes the timespan from the job submission time to the job completion time.

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

hand, *Coordinated Cold Data Migration* aims to migrate cold data for the initial input read phase of tasks for short jobs from hard disk to memory so as to shorten task runtime and resulting long task waiting time[2] under heavily-loaded periods. Eirene overlaps cold data migration for short tasks waiting in the queue on the worker nodes with the task waiting time, which is achieved by the coordination between distributed schedulers and worker nodes. On the other hand, *Scheduler-Aware Task Cloning* aims to duplicate every task of short jobs and use the result of the clones that are completed first under lightly-loaded periods. Eirene leverages the fact of tiny resource usage of short jobs and the availability of spare computing resources under light loads, and proactively launches extra copies of short tasks for straggler mitigation. Coordinated Cold Data Migration and Scheduler-Aware Task Cloning are tightly coupled into the general architecture of hybrid job schedulers and fully utilize the characteristics of distributed schedulers and worker nodes in hybrid job schedulers.

We implement a prototype of Eirene on top of Eagle [10], a state-of-the-art hybrid job scheduler. Experimental results demonstrate the effectiveness and efficiency of Eirene. For example, under heavy loads, Eirene is able to improve 50-percentile (P50), 75-percentile (P75), 90-percentile (P90) latency performance of short jobs by up to 44.4%, 80.3%, 84.1% respectively compared with Eagle under the Facebook trace with a cluster of 50000 nodes.

In summary, we make the following contributions as below:

- We propose the Coordinated Cold Data Migration scheme to migrate cold data for the initial input read phase of tasks for short jobs from hard disk to memory, so as to shorten task runtime of short jobs and resulting long task waiting time under heavily-loaded periods;
- We propose the Scheduler-Aware Task Cloning scheme to duplicate every task of short jobs and use the result of the clones that are completed first for straggler mitigation under lightly-loaded periods;
- We conduct extensive trace-driven experiments and validate the effectiveness of our proposed schemes.

II. Background and Motivation

A. Hybrid Job Schedulers

In order to leverage both high cluster utilization of centralized schedulers and fast scheduling decision making of distributed schedulers for the responsiveness of short jobs, hybrid job schedulers are proposed to combine one centralized scheduler and multiple distributed schedulers together. In general, a hybrid job scheduler, like Hawk [9], Eagle [10], Phoenix [11] and Dice [12], divides a cluster into two exclusive partitions: *general partition* and *short partition*, as shown in Figure 1. The short partition is dedicated to executing short jobs only while the general partition is used to execute both long and short jobs. The size of the short partition is determined by

[2]Task waiting time is the sum of task scheduling time and task queueing time. For hybrid job schedulers, task scheduling time is negligible because of multiple and parallel distributed schedulers. Thus task waiting time is dominated by task queueing time.

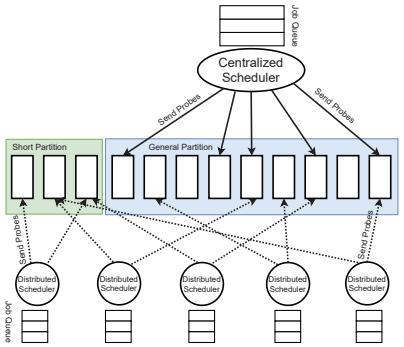


Fig. 1: A general architecture of hybrid job schedulers.

the resources consumed by short jobs, that is, the total task-seconds of short jobs (the sum of task runtime of tasks for all short jobs) over the total task-seconds of all jobs. Similar to YARN [6] and Mesos [7], the centralized scheduler in hybrid job schedulers is responsible for enqueueing and placing only long jobs onto worker nodes in the general partition. On the other hand, there are multiple distributed schedulers that can independently schedule only short jobs on any worker nodes in both partitions in parallel. Like Sparrow [8], distributed schedulers employ the "batch sampling" scheme to assign and enqueue a batch of task probes for short jobs into probe queues of randomly-chosen worker nodes. When a worker node becomes ready, it fetches one probe from its probe queue and then requests the executable package of one task from a distributed scheduler in charge of the corresponding job. When the worker node receives the task, it launches a container and executes the task program. Such immediate probe placement and late task assignment are also called "late binding".

On top of the general architecture of hybrid job schedulers, Hawk [9] introduces the "randomized task stealing" scheme, where idle worker nodes in the general partition steal task probes of short jobs behind running or waiting long tasks from randomly-chosen busy worker nodes, to compensate occasional poor scheduling decisions by distributed schedulers. Eagle [10] treats a probe as a proxy of the entire job instead of a single task and then proposes the "Sticky Batch Probing (SBP)" scheme. When a task is completed on a worker node, SBP continues to request and execute the remaining tasks of the job until all the tasks are executed. Further, Eagle mitigates the "head-of-line" blocking problem with the "Succinct State Sharing (SSS)" scheme, which shares the information about worker nodes where long jobs are either executing or waiting among distributed schedulers.

B. Experiment Study of Workload Fluctuation

To investigate the performance behaviors of short jobs under hybrid job schedulers, we conduct a trace-driven experiment study with the open-source Eagle simulator [14]. In the experiment, we simulate a cluster of 4000, 5000, 6000 worker nodes and then feed the Yahoo trace [1] as input workload to the simulator. In the Yahoo trace, 90.6% of the jobs are short

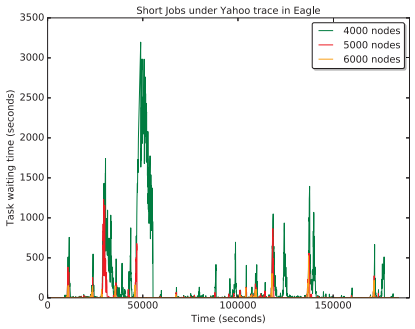


Fig. 2: Mean task waiting time of short jobs under the Yahoo trace in Eagle.

jobs and the total task-seconds of short jobs account for 2% of the overall task-seconds. As a result, 2% of the cluster is reserved for the short partition and the rest of the cluster is allocated for the general partition.

Figure 2 plots the mean task waiting time of short jobs under the Yahoo trace in Eagle. Task waiting time is defined as the duration between job submission time and the moment a task of the job begins execution. The reason we are interested in the metric of task waiting time of short jobs is we believe it is a good indicator of system loads and resource contentions. One can see from the figure that during most of the time the mean task waiting time of short jobs is close to 0, which means short tasks are almost immediately executed after their probes are put on worker nodes. This implies that the cluster is under light or moderate loads. On the other hand, we can see there are durations of extremely high mean task waiting time. For example, for the cluster of 4000 nodes, the mean task waiting time reaches a peak of over 3000 seconds. Such a long task waiting time clearly indicates the existence of heavy loads. Considering the mean task runtime of short jobs is less than 90.58 seconds, job latencies of short jobs are thus dominantly lagged by task waiting time. Even with the increase of worker nodes and resulting less intensity of workload, we can still see the lasting peaks of mean task waiting time. It is critically important to find a workable way to significantly reduce task waiting time and task runtime so as to lower latencies of short jobs. As workload fluctuation is evident from our study, we expect a desirable performance optimization approach for short jobs to be able to address performance issues under both lightly- and heavily-loaded periods.

C. Input Data Read Stage of Tasks and Cold Data Migration

For big data analytics jobs, the stage of reading input data usually accounts for a non-negligible portion of job latency. For example, reading map inputs of SQL queries on Hive takes up to 15% of query duration [15]. For another example, reading inputs from disk causes the first iteration of logistic regression jobs to run $15\times$ slower than late iterations [16]. Such a noticeable duration spent on accessing to singly-read data, that is, cold data, results from the fact that the input stage reads much more data than late stages after filtering and aggregation, while existing performance optimization approaches based on caching repeatedly/frequently-accessed data, like Spark [16], PACman [17], Triple-H [18], do not benefit the input data read stage of tasks.

To this end, Ignem [19] and DYRS [20] are two systems to migrate cold data from disk to memory before using them at the input read stage of task execution. Experiment results in Ignem show that reading input data from memory is $160\times$ faster than reading from hard disk, and Ignem improves hive queries by up to 34% [19]. The key to effective cold data migration is whether there is sufficient lead time, which is defined as the duration between job submission time and the moment the input data is accessed for a task. As we observe high task waiting time of short jobs under heavily-loaded periods, this becomes the best opportunity to overlap cold data migration with task waiting time of short jobs, which should in turn help reduce task runtime and improve latency performance of short jobs. However, it is very challenging to support cold data migration in the context of hybrid job schedulers because distributed schedulers themselves do not know which worker nodes will execute which tasks when they place probes of jobs onto randomly-chosen worker nodes due to batch sampling and late binding.

D. Straggler Problem and Mitigation

Stragglers, where one or more tasks of a job take much longer time to complete than other tasks, are commonly seen in enterprise production workloads. For example, more than 15% of straggler tasks for 25% phases are observed in a large cluster for the Bing search engine [21]. Moreover, we also analyze 4 representative traces derived from production workloads in enterprise data centers [1]–[4] and use the same definition of straggler tasks in Mantri research [21]: the tasks that take $1.5\times$ the median task runtime for a job. We find that in 3 out of these 4 traces, over 30% of short jobs have at least 1 straggler task as shown in Table I. There are many sources contributing to straggler tasks, like transient hardware errors or resource contentions, oversubscribed and congested networks, data skew in workloads (e.g., some tasks may take more input data than others due to imbalanced data distribution), Java just-in-time compilation overhead for the "first task" [15], [21]–[23].

As straggler problem is seen widespread, straggler tasks are also considered one major cause of lengthening job completion delay. For one example, job latency was lengthened by stragglers by 29% in Bing clusters [21]. For another example, straggler tasks could take up to $8\times$ longer than the mean task runtime in Hadoop clusters, causing the jobs to be slowed down by 47% on average [13]. As a result, a number of straggler mitigation approaches are proposed to address this issue [13], [15], [21]–[28], and the common strategy by most of them is *speculative execution*. Speculative execution spawns duplicate copies of straggler tasks when they are detected slow. The fundamental limitation of speculative execution is its hysteresis in response to straggler tasks. This

TABLE I: Stragglers in 4 traces of production workloads.

Trace	number of short jobs	portion of straggler jobs	number of short tasks	portion of straggler tasks
Yahoo	21,981	58.8%	514,583	10.7%
Cloudera	19,975	52.9%	3,897,480	4.4%
Google	455,891	4.4%	12,867,052	7.0%
Facebook	1,145,663	34.3%	11,724,548	8.2%

is because speculative execution needs to wait to monitor task progress and collect statistically sufficient samples to detect straggler tasks. More importantly, spawning redundant copies of straggler tasks at this point of time may be too late to be functional. To this end, Dolly [13] was proposed to completely avoid waiting and speculation by cloning every task of jobs and use the result of the clones that are completed first. The key to the effectiveness of task cloning like Dolly is whether there are sufficient spare computing resources in the cluster so that the duplicate copies of tasks can be launched nearly at the same time when the primary copies of tasks are started. Blindly applying Dolly's idea to hybrid job schedulers is impractical and could exacerbate job latency performance because it is resource-intensive to duplicate every task of both long and short jobs. This inspires us to consider cloning every task of short jobs by leveraging free resources under lightly-loaded periods, but it remains a challenging question of how to judiciously integrate task cloning into hybrid job schedulers.

III. Design and Implementation

A. Basic Idea

In this paper, we propose Eirene to improve latency performance of short jobs while minimizing adverse performance impact on long jobs in the context of hybrid job schedulers. More specifically, Eirene targets at improving job latency performance of data-parallel jobs like MapReduce jobs, where the input data is processed by the map tasks in parallel, and then feed to the reduce tasks after filtering and aggregation.

The basic idea behind Eirene is simple: Eirene performs cold data migration to shorten initial input read phase of tasks, and clones every task of short jobs to mitigate straggler tasks. Eirene is not simply applying Ignem [19] and Dolly [13] to hybrid job schedulers. Instead, Eirene leverages the ubiquitous workload fluctuation in enterprise data centers, and judiciously activates cold data migration for short tasks by leveraging long task waiting time during heavily-loaded periods, which is called *Coordinated Cold Data Migration*. On the other hand, Eirene duplicates every task for short jobs by exploiting free resources during lightly-loaded periods, which is called *Scheduler-Aware Task Cloning*. Note that both Coordinated Cold Data Migration and Scheduler-Aware Task Cloning are always enabled in Eirene, although they contribute to the job latency reduction under different load intensity. Eirene tightly couples these two functional modules into distributed schedulers and worker nodes of hybrid job schedulers, and significantly improve tail-latency performance of short jobs under fluctuating workloads.

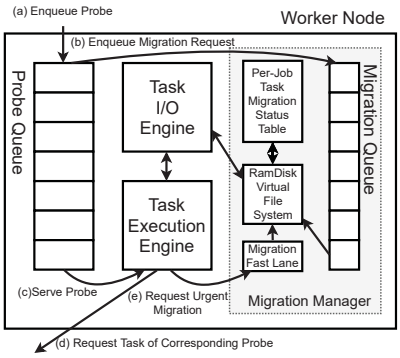


Fig. 3: An architectural diagram of a worker node with Coordinated Cold Data Migration support.

B. Coordinated Cold Data Migration

1) Architecture: Note that due to batch sampling and late binding, a distributed scheduler itself does not know which worker nodes will execute which tasks when it sends a batch of probes of a short job onto randomly-chosen worker nodes. This renders distributed schedulers incapable of dictating cold data migration solely. In Eirene, cold data migration is realized under the coordination between distributed schedulers and worker nodes. In the proposed Coordinated Cold Data Migration, worker nodes have delegated the autonomy of performing cold data migration in a distributed and parallel manner, while distributed schedulers are responsible for coordinating the data migration efforts of worker nodes and leveraging migrated data for accelerating initial input read phase of tasks.

Figure 3 depicts an architectural diagram of a worker node with Coordinated Cold Data Migration support. On the left side of the figure, there are Probe Queue (PQ), Task I/O Engine, and Task Execution Engine from the original worker node design in the general architecture of hybrid job schedulers. PQ is used to enqueue received probes from distributed schedulers. When a worker node becomes ready to execute a task, Task Execution Engine fetches one probe from PQ and requests a task of the corresponding job from the distributed scheduler. When it receives the task, Task Execution Engine launches a container and executes the task program within the container. If the task is involved with data reads or writes, Task I/O Engine is responsible for reading or writing data on the underlying distributed file system like HDFS (Hadoop File System). Coordinated Cold Data Migration augments a *Migration Manager* module, as shown on the right side of Figure 3. Migration Manager is meant to migrate cold data of short tasks from disks on local or remote nodes to local memory on the background. It is composed of 4 sub-modules: *Per-Job Task Migration Status Table*, *Migration Queue*, *Migration Fast Lane*, and *RamDisk Virtual File System*. In particular, Per-Job Task Migration Status Table (MST) is used to keep track of data migration progress for all waiting or running jobs on a worker node, including the information about job id, task number, migration

TABLE II: Per-Job Task Migration Status Table (MST).

job id	task no.	migration status	read by task?	location of migrated task data on RVFS
1	0	Not Migrated	N/A	N/A
1	1	Migrated	Yes	/ramdisk/job1_task1_data
1	2	Not Migrated	N/A	N/A
1	3	Migrating	No	/ramdisk/job1_task3_data

TABLE III: Per-Job Task Status Table (TST).

job id	task no.	task status	location of task data on distributed file systems like hdfs
1	0	Not Started	hdfs://node1:port/data/fileA_block0
1	1	Completed	hdfs://node2:port/data/fileA_block1
1	2	Running	hdfs://node3:port/data/fileA_block2
1	3	Not Started	hdfs://node4:port/data/fileA_block3

status, whether the migrated data is read, and the location of migrated task data on RVFS, as one example MST table shown in Table II. Migration Queue (MQ) enqueues migration requests associated with probes staying in Probe Queue, and its function is to enforce I/O bandwidth management of data migration to avoid contentions on foreground task execution. Migration Fast Lane (MFL) is used to accommodate urgent data migration requests without delays, which is important to allow Coordinated Cold Data Migration to collaborate with the Sticky Batch Probing (SBP) feature in Eagle [10]. RamDisk Virtual File System (RVFS) is responsible for storing migrated data in RamDisk and servicing read requests from the initial phase of tasks. Moreover, RVFS leverages the MST information to evict migrated data and reclaim space for new task data being migrated.

On the other hand, distributed schedulers in Coordinated Cold Data Migration maintain Per-Job Task Status Table (TST) to track the task progress of jobs as well as the location of the input data on the underlying distributed file system for all the tasks of every short job, as one example TST table is shown in Table III. More importantly, distributed schedulers augment every probe with the TST information upon placing them onto worker nodes.

2) Workflow: In Coordinated Cold Data Migration, worker nodes have delegated autonomy of decision making of data migration, that is, a worker node itself decides whether and which tasks for data migration given a probe in its Probe Queue (PQ). When a worker node receives a probe of a job containing the TST table and enqueues the probe in PQ, Eirene generates a predefined number of random task numbers and enqueues migration requests of them into the MQ queue. On the other hand, when a worker node receives the response from the distributed scheduler about the next task of a job to execute accompanied with the TST table, the worker node will examine both TST and MST tables to see if there is any task of the same job, which is not started and whose data is not migrated. If yes, the worker node will put the task number into MFL, and perform data migration immediately. By doing so, the worker node is able to overlap the execution of the current task with data migration for the next task to execute, aligned with the "Sticky Batch Probing (SBP)" scheme in Eagle.

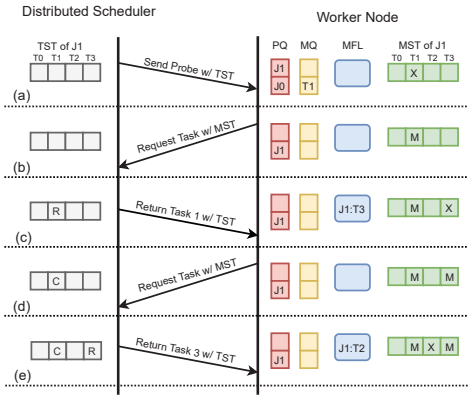


Fig. 4: An example of the interaction between the distributed scheduler and worker node. J0 and J1 denote jobs 0 and 1 respectively. T0-T3 denote tasks 0 to 3 of job 1 respectively. In TST, "S", "R", "C" denote "Scheduled", "Running", and "Completed" respectively. In MST, "M" and "X" denote "Migrated" and "Migrating" respectively.

At the side of the distributed scheduler, it reads the MST table embedded in the request of the next task by the worker node and thus knows which tasks have data already migrated to memory. Then it chooses one random task from the tasks that are not started but whose data have been migrated to memory, and responds to the worker node with the chosen task number. If none of such tasks are found, the distributed scheduler just returns any "Not Started" task number to the worker node. By doing so, the coordination between a distributed scheduler and a worker node is able to maximize potential performance gains from cold data migration.

Figure 4 illustrates an example of the interaction and coordination between a distributed scheduler and a worker node in Coordinated Cold Data Migration. Figure 4(a) shows a point-in-time system snapshot *after* a distributed scheduler receives a new job, say Job 1 (J1), and sends one probe containing the TST table to a worker node. At the time, the worker node is executing one task of Job 0 (J0), as we can see that a probe of J0 is staying at the head of PQ, followed by the probe of J1. The worker node reads the TST table and knows none of J1's tasks are completed or running, so T1 of J1 is randomly chosen and enqueued into MQ for data migration. Because of no ongoing data migration, the worker node starts to migrate data for T1 immediately, as T1 is marked "migrating" in the MST table. Figure 4(b) shows a system snapshot *before* the worker node requests the next task to execute from the distributed scheduler, while T1 is shown to be migrated already. Figure 4(c) shows that the distributed scheduler returns T1 to the worker node. The worker node then executes T1 and starts to migrate data for T3 immediately. As shown in Figure 4(d), the worker node finishes the execution of T1 and data migration for T3, and requests the next task from the distributed scheduler again. Figure 4(e) shows that the distributed scheduler responds to the worker node with T3,

and the worker node then starts to execute T3 and performs data migration for T2.

3) Implementation Issues: We discuss a few key implementation issues in the below:

- *How to enforce I/O bandwidth management with MQ?* Migration Manager specifies the maximum number of allowed concurrent migrations (CM) to limit the maximum disk bandwidth for data migration. Note that experiment results in Ignem [19] show that it takes 6.42 seconds on average to read an HDFS block of 64MiB from hard disk to memory. Assuming CM is 10, then disk bandwidth used for data migration is $10 \times \frac{64}{6.42} = 99.6$ MiB/s at the peak, which is lower than sustainable sequential read throughputs of hard disks on the market. When the current number of concurrent migrations reaches the CM threshold, the additional migration requests will be enqueued and waiting in MQ.
- *Is there sufficient memory to store migrated data?* One possible concern is whether the memory capacity of a worker node is sufficient to store the migrated data so as to avoid the case of the migrated data being evicted from memory before it is used. From the specification information of Amazon AWS EC2 Instance Types [29], we know the ratio of the number of vCPUs to memory capacity (in GiB) ranges from 1:2 to 1:12. For a worst-case analysis, assuming a vCPU can run 2 tasks in parallel, an EC2 instance with 96 vCPUs needs $96 \times 2 \times 64$MiB = 12GiB memory to store migrated data, which is far less than the memory capacity of the instance: $96 \times 2 = 192$GiB. Analysis from Ignem [19] and PACman [17] also confirm that there is sufficient memory to store migrated data.
- *How to speed up reading input with migrated data on RVFS?* Remember that the request of the next task to execute sent by the worker node contains the MST table, which includes the location of migrated data on RVFS. Therefore, the distributed scheduler leverages such information and passes the location of migrated data to the task. Moreover, a small modification to the task program is made to read migrated data from RVFS if available, or otherwise read data from the original location on the underlying file system. A simplified version of the example code snippet to read input data is shown in Listing 1.

Listing 1: Example Python code snippet of reading input data.

```
import os

# assume HDFS is used as the underlying distributed file system.
# assume hdfsRead() and rvfsRead() are provided APIs to read data
# from HDFS and RVFS respectively.
def readInputData(location_on_hdfs):
    # if environment variable of data location on rvfs exists.
    if os.environ['location_on_rvfs']:
        # read data from rvfs
        dataBuffer = rvfsRead(os.environ['location_on_rvfs'])
    else:
        # read data from HDFS
        dataBuffer = hdfsRead(location_on_hdfs)

    return dataBuffer
```

C. Scheduler-Aware Task Cloning

In order to effectively mitigate stragglers and improve latencies of short jobs, Scheduler-Aware Task Cloning in Eirene duplicates every task of short jobs and uses the results of the tasks that finish first, the same as Dolly [13]. However, Scheduler-Aware Task Cloning distinguishes itself from the existing approaches like Dolly in two important aspects as below:

- *Short Jobs Oriented*. Scheduler-Aware Task Cloning is applied to short jobs only. This is because Eirene recognizes the fact in production environments short jobs consume a very small portion of resources but they are also latency-sensitive. Replicating all tasks of short jobs will not likely result in resource contentions. More importantly, Scheduler-Aware Task Cloning leverages the fluctuating nature of workloads and activates task cloning only under idle or lightly-loaded periods.
- *Distributed Scheduler Aware*. Scheduler-Aware Task Cloning intentionally minimizes changes to hybrid job schedulers for simplicity and feasibility. As a result, it is designed to be tightly coupled with a distributed scheduler to leverage its inherent feature of "batch sampling".

In detail, for a given short job consisting of N tasks, the original batch-sampling scheme sends probes to $2 \times N$ randomly-chosen worker nodes. After the N tasks are assigned to the first N probes whose worker nodes request the distributed scheduler of tasks to execute, the later N probes are canceled when the corresponding worker nodes request tasks to execute from the distributed scheduler. Then the worker nodes fetch the next probes and may execute tasks of other jobs, like the example shown in Figure 5(a). In contrast, Scheduler-Aware Task Cloning tries to leverage such probes and repurpose them to represent clones of tasks that have not been completed. In addition to the "task status" column in Per-Job Task Status Table, Scheduler-Aware Task Cloning adds one new column, called "cloned task status", to track the status of cloned tasks. When the distributed scheduler receives a request of a task to execute from a worker node, it checks if all the primary copies of tasks have been launched or completed ("Started" and "Completed" status in the "task status" column). If yes, the distributed scheduler will try to find a running task that has no clone (from the "cloned task status" column) and return its task number to the worker node. The worker node does not differentiate a primary copy or a duplicate copy of a task. It just launches a container and executes the received task program from the distributed scheduler. The distributed scheduler may receive two task completion messages, but it marks the task as "Completed" in the table only when the first one arrives and simply ignores the second one. An example timeline of task scheduling and execution with Scheduler-Aware Task Cloning is shown in Figure 5(b).

It has to be noted that Scheduler-Aware Task Cloning also cooperates with Coordinated Cold Data Migration for maximizing the performance potential of data migration.

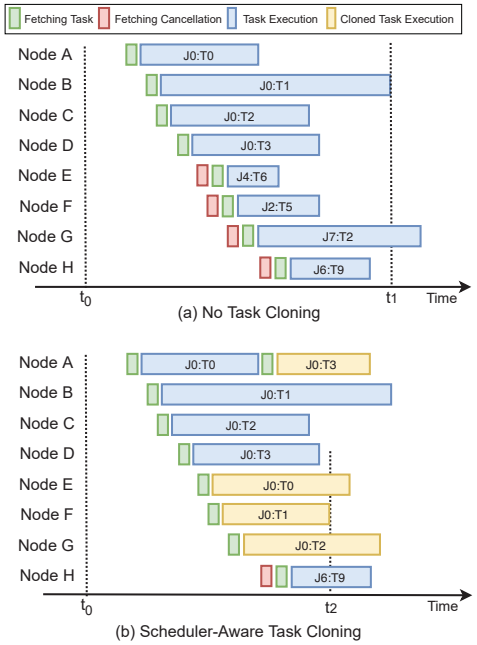


Fig. 5: An example with and without Scheduler-Aware Task Cloning. Figure (a) shows the timeline of the task execution of a 4-task job ("J0") under the original batch sampling scheme. t_0 denotes the time when J0 is submitted to the job scheduler. t_1 denotes the completion time of T1, which is the straggler task. Job latency of J0 is thus $t_1 - t_0$. Figure (b) shows the timeline of the task execution of J0 with Scheduler-Aware Task Cloning. Note that the probes on worker nodes E - G are repurposed to represent and execute cloned tasks. Because of Sticky Batch Probing, the probe on worker node A is also repurposed to represent and execute the last clone task after T0 is completed. T1's clone completes at t_2, so the job latency of J0 becomes $t_2 - t_0$, which is shorter than $t_1 - t_0$.

In determining which task to clone, Scheduler-Aware Task Cloning gives the preference to the worker nodes that have migrated data of tasks that have no clones yet. However, we expect the chance of Coordinated Cold Data Migration and Scheduler-Aware Task Cloning being concurrently activated is low because they are usually activated under different load intensity conditions.

IV. Performance Evaluations

A. Experimental Setup

In order to evaluate the effectiveness and efficiency of the proposed Coordinated Cold Data Migration and Scheduler-Aware Task Cloning schemes, we implement a prototype of Eirene on top of the state-of-the-art hybrid job scheduler, Eagle [10] in the open-source Hawk/Eagle simulator [14], which is widely used in research work of Sparrow [8], Hawk [9], Eagle [10], Phoenix [11], Dice [12].

We feed the same traces used in Section II-D as input workload to the simulator, and the detailed characteristics of traces are shown in Table IV. The original Facebook trace is long, we just use the first 100,000 records in the experiments to reduce simulation time. The jobs with mean task runtime less than cutoff task runtime are short jobs. The ratio of total task-seconds of short jobs to those of all jobs also dictates the size of the short partition.

TABLE IV: Trace characteristics.

Trace	Total jobs	Cutoff task runtime	Short jobs	Task-seconds of short jobs
Yahoo	24,262	90.6 seconds	90.6%	2%
Cloudera	21,030	272.8 seconds	95.0%	9%
Google	506,546	1129.5 seconds	90.0%	17%
Facebook	100,000	76.6 seconds	98.0%	2%

TABLE V: Configuration parameters of simulations.

Description	Abbr.	Values
Coordinated Cold Data Migration (CCDM)		
Max. number of concurrent migrations	CM	10
Max. number of migrations per probe	MPP	2
Data migration time (seconds)	DMT	6.42
Scheduler-Aware Task Cloning (SATC)		
Projected runtime of cloned tasks	PRT	random
Cluster scale	**Values**	
Cluster sizes for the Yahoo trace	3500, 4000, 4500, 5000	
Cluster sizes for the Cloudera trace	13000, 13500, 14000, 14500	
Cluster sizes for the Google trace	12000, 13000, 14000, 15000	
Cluster sizes for the Facebook trace	50000, 55000, 60000, 65000	

Table V shows the configuration parameters of simulations. Note that MPP denotes the maximum number of randomly-generated task numbers to migrate per probe when a probe is placed into the PQ queue by the distributed scheduler. Data migration time of 6.42 seconds is cited from the Ignem research [19], and this is the estimated amount of task runtime reduction for the tasks whose input data has been migrated to RVFS and then directly used by the tasks. Regarding the Scheduler-Aware Task Cloning scheme, since the traces do not include task runtime information for cloned tasks, we use runtime of randomly-chosen primary copies of tasks of the same job to project runtime of cloned tasks. Table V also gives the configurations of cluster size used in our experiments. We vary cluster sizes to study the scalability of Eirene. In particular, the cluster sizes are carefully chosen to demonstrate the performance trend from the overloaded case to the moderately-loaded case. We run simulations with bigger cluster sizes but the experiment results are consistent with the trend, so we omit to report most of their results in the paper.

Regarding the performance metrics, we consider 50-percentile (P50), 75-percentile (P75), 90-percentile (P90) job latencies as key metrics to evaluate the latency performance of short jobs. Moreover, we focus on normalized performance numbers, which are the ratio of P50, P75, P90 latency numbers from Eirene to the ones from the original Eagle respectively.

B. Results

1) Performance Analysis: Figure 6 depicts the normalized latency performance of Eirene compared with Eagle as a

function of different cluster sizes, under the four traces. We obtain a few observations from the figure. First, we clearly see the consistent performance improvement by Eirene across the 4 different workloads. Second, significant performance improvement is observed under the overloaded case. For example, Eirene improves P50, P75, P90 latency performance of short jobs by up to 44.4%, 80.3%, 84.1% respectively compared with Eagle, under the Facebook trace with a cluster of 50000 nodes. It is clear that Eirene is able to drastically improve the latency performance of short jobs by shortening task waiting time and resulting long latencies of jobs under the Yahoo and Cloudera traces as well. Under the Google trace, Eirene achieves 5.9%, 10.9%, 8.9% performance improvement in terms of P50, P75, P90 latency. This is because the cutoff task runtime of short vs. long tasks for the Google trace is 1129.5 seconds, which is 175 times longer than the estimated task runtime reduction of 6.42 seconds if the input data is migrated and accessed locally. This implies some correlation between the extent of performance improvement and the ratio of task runtime reduction brought by Eirene to the task runtime of short jobs. Third, we can observe that the performance improvement is decreased with the increase of cluster size. It means that Eirene benefits heavy workloads more than light workloads. In summary, Eirene is shown to consistently improve job-latency performance of short jobs across different cluster sizes in a scalable manner.

To understand the contributions to performance improvement by Coordinated Cold Data Migration (CCDM thereafter) and Scheduler-Aware Task Cloning (SATC thereafter) individually, we conduct experiments with the two schemes in overloaded and lightly-loaded cases. Figure 7 plots the P50, P75, P90 latency performance of short jobs normalized to Eagle in the overloaded case with Eagle, CCDM, SATC, and Eirene. One can see that in the overloaded case, CCDM is the sole contributor to performance improvement compared with Eagle across all the four traces, while SATC does not result in any performance improvement. This is expected because SATC is not actually activated if the cluster is kept overloaded, while the high task waiting time due to overloading is leveraged by CCDM to migrate cold data for task runtime reduction.

On the other side, Figure 8 plots the P50, P75, P90 latency performance of short jobs normalized to Eagle in the lightly-loaded case with different schemes. One can see that in this case, the performance improvement results from the SATC scheme under the Yahoo and Facebook traces. For example, SATC improves the P50, P75, P90 latency performance of short jobs of Eagle by 11.4%, 7.2%, and 4.8% respectively under the Facebook trace. This is expected since abundant computing resources under light loads enable SATC to execute cloned copies of tasks of short jobs nearly at the same time the primary copies of tasks are executed. We have two additional observations. First, we can see trivial performance improvement for the Cloudera trace. It is likely because this trace has only 4.4% straggler tasks over all of the tasks for short jobs. Second, we notice there is a slight performance regression (up to -3.1% on P50 latency) under the Google trace for SATC since this trace has only 4.4% straggler jobs. In addition, short jobs in the Google trace have a relatively large task cutoff runtime (1129.5 seconds), which may cause the task cloning scheme to consume considerable computing resources.

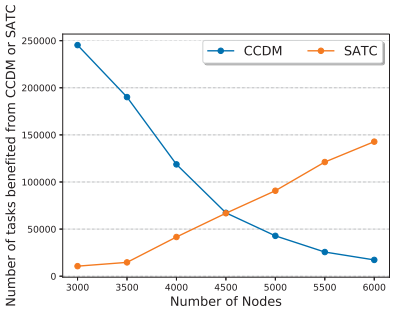
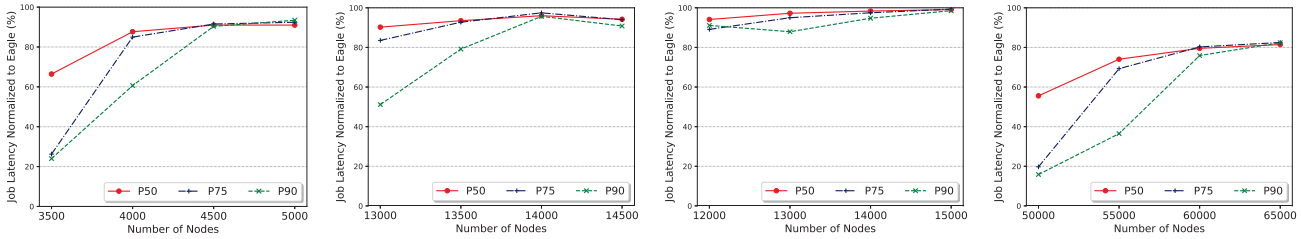


Fig. 9: The number of short tasks benefited from CCDM or SATC under the Yahoo trace as the cluster scale increases.

We add two counters to the simulator to keep track of the total number of short tasks benefited from CCDM and SATC respectively. The first counter is incremented for every occurrence when a task reads input data from RVFS rather than the underlying distributed file system. The second counter is incremented for every occurrence when the cloned copy of a task completes earlier than the primary copy. To illustrate the trend of contributions to performance improvement by CCDM and SATC as the increase of cluster size, we plot the trend of these two counters under the Yahoo trace on a cluster of from 3000 nodes to 6000 nodes. We can see from Figure 9, as cluster size increases, the number of tasks benefited from CCDM decreases linearly with a steep slope and the number of tasks benefited from SATC increases linearly with a slow slope. In addition to the fact that tail latencies of short jobs are mainly affected under heavily-loaded periods rather than lightly-loaded periods, this may be another reason why SATC obtains fewer performance gains than CCDM and we see decreasing performance improvement as the cluster scale increases in Eirene.

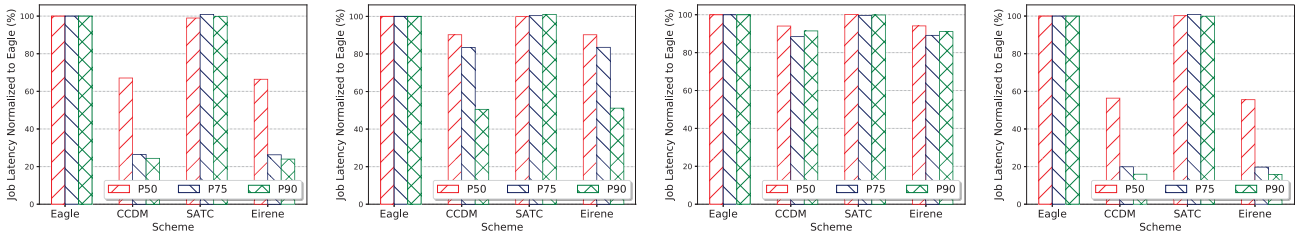
2) Sensitivity Study: As we witness from the above section that CCDM plays a more important role than SATC in terms of the latency performance of short jobs, it is desirable to quantitatively evaluate the impact of tunable parameters of CCDM shown in Table V. In the following, we vary two key CCDM configuration parameters and evaluate their performance impacts under the Yahoo trace on a cluster of 4000 nodes as a case study.

Data migration time (DMT). Figure 10 plots the latency performance and the number of tasks benefited from CCDM with the varying DMTs to simulate different disk/network speeds. It is reasonable to see the number of tasks benefited from CCDM drops as the increase of DMT because a longer DMT has a higher chance to miss more opportunities for performing and benefiting from data migration. However, we find that the latency performance is actually improved with



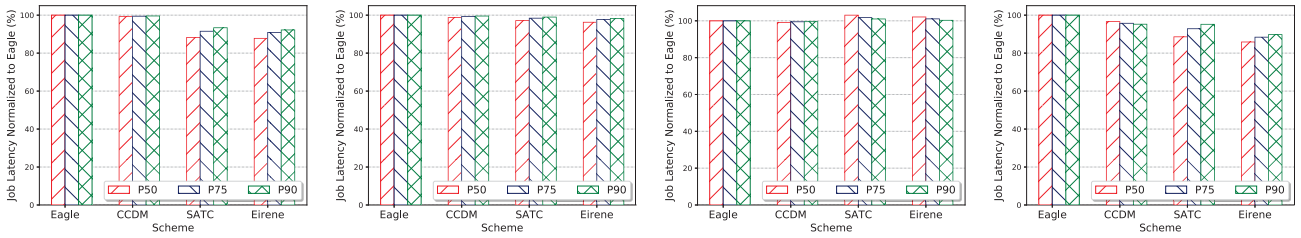
(a) Yahoo trace (b) Cloudera trace (c) Google trace (d) Facebook trace

Fig. 6: P50, P75, P90 latency performance of short jobs normalized to Eagle with different cluster sizes.



(a) Yahoo trace, 3500 nodes (b) Cloudera trace, 13000 nodes (c) Google trace, 12000 nodes (d) Facebook trace, 50000 nodes

Fig. 7: P50, P75, P90 latency performance of short jobs normalized to Eagle with different schemes in the overloaded case.



(a) Yahoo trace, 6000 nodes (b) Cloudera trace, 18000 nodes (c) Google trace, 20000 nodes (d) Facebook trace, 90000 nodes

Fig. 8: P50, P75, P90 latency performance of short jobs normalized to Eagle with different schemes in the lightly-loaded case.

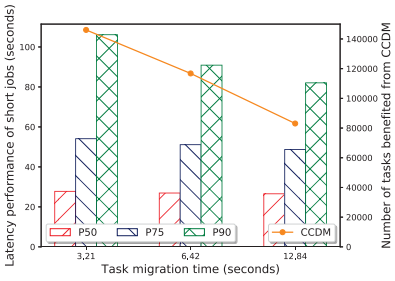


Fig. 10: Latency performance and the number of tasks benefited from CCDM with different DMTs.

longer DMT. One possible reason is that, with the doubling of the data migration time (that is also the estimated task runtime reduction), the number of tasks benefited from CCDM is decreased linearly. The accumulated performance gain from longer DMT outweighs the accumulated performance loss from the decrease of benefited tasks.

Maximum number of concurrent migrations (CM). Figure 11 plots the latency performance and the number of tasks benefited from CCDM with the varying CMs. We can see that there are almost no noticeable changes to latency performance and the total number of tasks benefited from CCDM with the increase of CM from 5 to 20. This indicates that migration bandwidth is not a performance bottleneck, and we can choose a conservative value for CM.

V. Extended Related Work

Performance Optimizations for Hybrid Job Schedulers. In addition to the aforementioned Hawk [9] and Eagle [10] schedulers, the most relevant work to Eirene is Dice [12]. Dice is also meant to improve job latency performance of short jobs under hybrid job schedulers. In Dice, "Elastic Sizing" aims to dynamically adjust the short partition size so as to prioritize short jobs over long jobs under heavy loads, while "Opportunistic Preemption" aims to opportunistically preempt resources of running long tasks in the general partition so as to mitigate the head-of-line problem by short tasks on

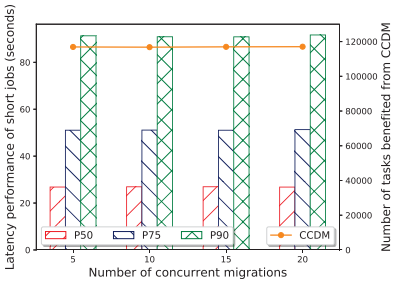


Fig. 11: Latency performance and number of tasks benefited from CCDM with different CMs.

the same worker nodes. In contrast, Eirene tackles different problems and takes a totally different approach. Coordinated Cold Data Migration focuses on shortening input data reading time of short jobs so as to minimize both task runtime and resulting long task waiting time of short jobs. In the meantime, Scheduler-Aware Task Cloning addresses the straggler problem and proactively clone tasks under light loads. Note that the proposed schemes in Eirene are orthogonal to Dice, so both approaches can be easily integrated to maximize the performance gains.

Cold Data Migration. In addition to the aforementioned Ignem [19] and DYRS [20] cold data migration systems, the most relevant work to Coordinated Cold Data Migration is HPMR [30] and MEMTUNE [31]. In HPMR, an inter-block prefetching scheme prefetches the HDFS blocks from a remote rack to the server that is supposed to process such blocks. Similarly, MEMTUNE is proposed to exploit the DAG execution graph of Spark jobs and prefetch data that will be used by the next stages. Different from these prefetching schemes in centralized job schedulers, Coordinated Cold Data Migration enables data prefetching for short jobs in hybrid job schedulers whose distributed schedulers are incapable of dictating data migration solely. This is achieved by the carefully-designed coordination between distributed schedulers and worker nodes.

VI. Conclusion

In this paper, we propose Eirene to improve the job latency performance of short jobs for hybrid job schedulers under fluctuating workloads. In Eirene, Coordinated Cold Data Migration judiciously leverages long task waiting time of short jobs during heavily-loaded periods and performs cold data migration under the coordination between distributed schedulers and worker nodes to shorten the time of reading input data at the initial stage of tasks. Furthermore, Scheduler-Aware Task Cloning proactively clones every task of short jobs during lightly-loaded periods to address the straggler problem. Experiment results from a prototype implementation of Eirene on top of a state-of-the-art hybrid job scheduler demonstrate the effectiveness and efficiency of the proposed schemes.

Acknowledgment

We thank the anonymous reviewers for their constructive suggestions on this paper. This research has been partially supported by the National Science Foundation through grants IIS-1423487, ICER-1541043, and IIS-1652846. The contents do not necessarily reflect the views and policies of the funding agencies and do not mention of trade names or commercial products constitute endorsement or recommendation for use.

References

[1] Yanpei Chen et al. The case for evaluating mapreduce performance using workload suites. In *MASCOTS '11*, 2011.
[2] Yanpei Chen et al. Interactive query processing in big data systems: A cross-industry study of mapreduce workloads. In *VLDB '12*, 2012.
[3] Charles Reiss et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC '12*, 2012.
[4] Bikash Sharma et al. Modeling and synthesizing task placement constraints in google compute clusters. In *SoCC '11*, 2011.
[5] George Amvrosiadis et al. On the diversity of cluster workloads and its impact on research results. In *USENIX ATC '18*, 2018.
[6] Vinod Kumar Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC '13*, 2013.
[7] Benjamin Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI '11*, 2011.
[8] Kay Ousterhout et al. Sparrow: Distributed, low latency scheduling. In *SOSP '13*, 2013.
[9] Pamela Delgado et al. Hawk: Hybrid datacenter scheduling. In *USENIX ATC '15*, 2015.
[10] Pamela Delgado et al. Job-aware scheduling in eagle: Divide and stick to your probes. In *SoCC '16*, 2016.
[11] Prashanth Thinakaran et al. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *ICDCS '17*, 2017.
[12] Wei Zhou et al. Improving short job latency performance in hybrid job schedulers with dice. In *ICPP '19*, 2019.
[13] Ganesh Ananthanarayanan et al. Effective straggler mitigation: Attack of the clones. In *NSDI '13*, 2013.
[14] Pamela Delgado. Hawk/eagle simulator. https://github.com/epfl-labos/eagle/tree/master/simulation, 2017.
[15] Kay Ousterhout et al. Making sense of performance in data analytics frameworks. In *NSDI 15*, 2015.
[16] Matei Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI '12*, 2012.
[17] Ganesh Ananthanarayanan et al. Pacman: Coordinated memory caching for parallel jobs. In *NSDI '12*, 2012.
[18] Nusrat Sharmin Islam et al. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *CCGrid '15*, 2015.
[19] Simbarashe Dzinamarira et al. Ignem: Upward migration of cold data in big data file systems. In *ICDCS '18*, 2018.
[20] Simbarashe Dzinamarira et al. Dyrs: Bandwidth-aware disk-to-memory migration of cold data in big-data file systems. In *IPDPS '19*, 2019.
[21] Ganesh Ananthanarayanan et al. Reining in the outliers in map-reduce clusters using mantri. In *OSDI '10*, 2010.
[22] YongChul Kwon et al. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD '12*, 2012.
[23] Ganesh Ananthanarayanan et al. Scarlett: Coping with skewed popularity content in mapreduce clusters. In *EuroSys '11*, 2011.
[24] Matei Zaharia et al. Improving mapreduce performance in heterogeneous environments. In *OSDI '08*, 2008.
[25] Xiaoqi Ren et al. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *SIGCOMM '15*, 2015.
[26] Ganesh Ananthanarayanan et al. Grass: Trimming stragglers in approximation analytics. In *NSDI '14*, 2014.
[27] Neeraja J. Yadwadkar et al. Wrangler: Predictable and faster jobs using fewer resources. In *SoCC '14*, 2014.
[28] Chen Chen et al. Speculative slot reservation: Enforcing service isolation for dependent data-parallel computations. In *ICDCS '17*, 2017.
[29] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/.
[30] Sangwon Seo et al. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *IEEE International Conference on Cluster Computing and Workshops*, 2009.
[31] Luna Xu et al. Memtune: Dynamic memory management for in-memory data analytic platforms. In *IPDPS '16*, 2016.