Fregata: A Low-Latency and Resource-Efficient Scheduling for Heterogeneous Jobs in Clouds

Jinwei Liu
Department of Computer and Information Sciences
Florida A&M University, Tallahassee, FL 32307, USA
jinwei.liu@famu.edu

***Abstract*—An increasing number of large-scale data analytics frameworks move towards larger degrees of parallelism aiming at low-latency guarantees. It is challenging to design a scheduler with low latency and high resource utilization due to task dependency and job heterogeneity. The state-of-the-art schedulers in cloud/datacenters cannot well handle the scheduling of heterogeneous jobs with dependency constraints (e.g., dependency among tasks of a job) for simultaneously achieving low latency and high resource utilization. The key issues lie in the scalability in centralized schedulers, ineffective and inefficient probing and resource sharing in both distributed and hybrid schedulers. To address this challenge, we propose Fregata, a low-latency and resource-efficient scheduling for heterogeneous jobs with constraints (e.g., dependency constraints among tasks of a job) in clouds. Fregata first uses the machine learning algorithm to classify jobs into two categories (high priority jobs and low priority jobs) based on the extracted features. Next, Fregata splits the jobs into tasks and distributes the tasks to the master nodes based on task dependency and the load of master nodes. Then, Fregata utilizes the dependency information of tasks to determine task priority (tasks with more dependent tasks have higher priority), and packs tasks by leveraging the complementary of tasks' requirements on different resource types and task dependency. Finally, the master nodes distribute tasks to workers in the system based on priority of tasks and workers and the resource demands of tasks and the available resources of workers. To test the performance of Fregata, we conduct trace-driven experiments. Extensive experimental results based on a real cluster and Amazon EC2 cloud service show that Fregata achieves low-latency and high resource utilization compared to existing schedulers.**

***Index Terms*—scheduling, task dependency, resource utilization, latency, machine learning**

I. Introduction

Production data-parallel jobs in cloud increasingly have complex dependencies in computation [1]. Modern data analytics frameworks compile programs into job DAGs (directed acyclic graphs) consisting of many dependent tasks [1]. The dependency constraints of job DAGs pose significant challenges to job/task scheduling, making it difficult to balance common objectives such as high resource utilization, fast job completion and fair sharing [1]. Figure 1 shows an example of diverse dependencies among tasks of production data-parallel jobs in cloud. In Figure 1, tasks T_1, T_5 and T_{15} are precedent tasks, and their dependent tasks (e.g., tasks T_2, T_6 , T_7, T_8, T_9, T_{16}, T_{17}, T_{18}) cannot start execution until they finish execution. Inappropriate execution order of tasks may drag down the system performance, but judicious selection of

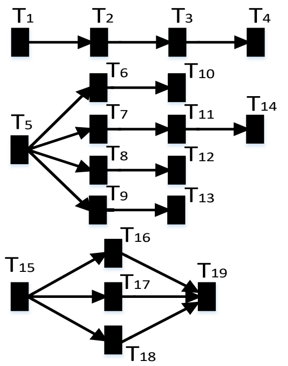


Fig. 1: Diverse dependencies among tasks of production data-parallel jobs.

execution order of tasks can help reduce the latency. Task T_5 has more dependent tasks than tasks T_1 and T_{15}. Executing T_5 at first can enable more dependent tasks to start executing after the precedent task T_5 completes, which helps reduce the latency.

Workload heterogeneity has been a long-standing challenge in datacenter scheduling [2]–[4]. Modern datacenters also encounter increasingly heterogeneous workloads composed of long batch jobs such as data analytics, and latency-sensitive short jobs such as operations of user-facing services [2]–[5]. Durations of tasks in Microsoft production clusters can vary from a few milliseconds to tens of thousands of seconds [5], and a significant fraction of tasks are short-lived (∼50% last less than 10 seconds). Jobs with diverse execution time and fanout degree (number of tasks of a job) pose distinct requirements for scheduling. Short jobs are sensitive to scheduling delays and typically have higher priority in scheduling. Long jobs usually have larger fanout and can tolerate some scheduling delays, and thus usually have lower priority; but long jobs typically have high resource demands and require high-quality scheduling, e.g., improving resource utilization and load balance. The job heterogeneity not only challenges the performance improvement on latency and throughput, but also poses a challenge for improving resource utilization. Figure 2 shows an example of job heterogeneity in execution time and resource requirements in cloud datacenters, and the resource wastage caused by resource fragmentation can easily occur if a worker is allocated to Jobs 1 and 2. The scheduler has to take care to avoid problems such as head-of-line

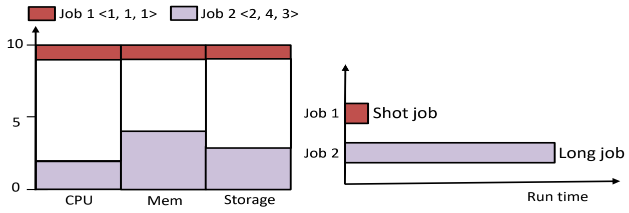


Fig. 2: An example of job heterogeneity in cloud datacenters.

blocking (especially under high load) and resource wastage (e.g., resource wastage caused by resource fragmentation). Therefore, it is common practice to collocate short jobs and long jobs in datacenter scheduling, but simultaneously meeting the diverse needs of heterogeneous jobs remains a critical challenge [2], [6], [7].

Early job schedulers such as Jockey [8], Quincy [9], Tetrished [10], Delay Scheduling [11], Firmament [12] and Yarn [13] are centralized schedulers. Centralized schedulers require a global view of resource availability to make scheduling decisions. As systems scale, a very large number of scheduling decisions and status reports from a large number of servers can overwhelm centralized schedulers, which becomes a bottleneck and results in significant scheduling delay for jobs. This is especially problematic for short jobs that are typically latency-bound. Recently, distributed schedulers such as Sparrow [14], Peacock [15], Pigeon [2] and hybrid schedulers such as Mercury [16], Hawk [17] and Eagle [4] have been proposed to overcome the limitation of the centralized schedulers. However, distributed schedulers typically have non-negligible probe processing overheads due to maintaining a fairly large amount of probe related states and lack coordination among one another for enforcing global service differentiation among jobs; hybrid schedulers have difficulty in mitigating the negative impact of long jobs on the performance of short jobs. Also, the above centralized schedulers and hybrid schedulers cannot well handle the scheduling of heterogeneous jobs with dependency constraints for simultaneously achieving low latency and high resource utilization.

To address the problem, in this paper, we aim to develop Fregata: a dependency-aware and resource efficient scheduling for heterogeneous jobs in clouds. Our goal is to achieve low latency and high resource utilization simultaneously. Fregata outperforms previous schedulers in that it can well handle the scheduling of heterogeneous jobs with dependency constraints, and simultaneously achieve low latency and high resource utilization by leveraging task dependency to determine task priority and the complementary of tasks' requirements on different resource types for reducing resource fragmentation. We summarize the contributions of this work below.

- We propose Fregata, a dependency-aware and resource efficient scheduling for heterogeneous jobs, which can reduce the latency and improve resource utilization in clouds.
- We consider job heterogeneity and propose a machine learning based method for job classification based on the extracted features.

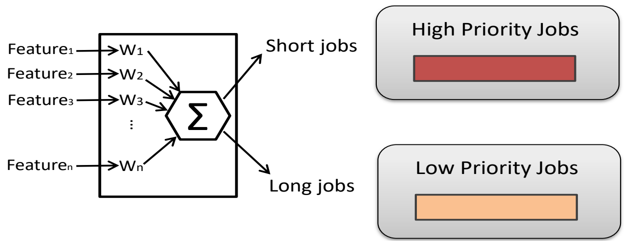


Fig. 3: Job classification and priority determination.

- Fregata splits jobs into tasks by taking into account the constraints (e.g., the dependency among tasks) of tasks, and assigns tasks that do not depend on each other to different workers (or different cores of a worker) so that these tasks can be processed in parallel and the latency can thus be reduced. Fregata leverages task dependency to determine task priority and adequately considers task dependency in scheduling to further reduce latency.
- We adequately consider the diverse demands of tasks, and propose a task packing strategy to reduce the resource fragmentation and improve the resource utilization.

The remainder of this paper is organized as follows. Section II describes the system model used in this paper. Section III presents the design of our proposed job scheduling Fregata. Section IV presents the performance evaluation for Fregata. Section V reviews the related work. Section VI concludes this paper with remarks on our future work.

II. System Model

In this section, we first introduce some concepts and assumptions, then we formulate our problem based on the concepts and assumptions. Finally, we present our proposed algorithm for reducing latency and improving resource utilization with the consideration of task dependency.

A. *Concepts and Assumptions*

In a distributed system, there are workers and schedulers. Workers are used to run tasks. A job, typically consists of a set of tasks that can run in parallel on different workers (or different cores of a worker), is supposed to be split into m tasks, and the tasks are allocated to workers. Scheduling a job requires assigning every task of a job to a worker. We assume jobs can be handled by any scheduler and tasks are run by workers. Each worker has a buffer queue which is used for queueing tasks when a worker is allocated to more tasks than it can run concurrently. When a new task is scheduled on an idle worker, the task starts executing immediately. The completion time of a task is the time from the submission of the job that contains the task to the time when the task finishes execution. A job completes when all of its tasks finish.

B. *The Optimization of the Resource Utilization and Latency*

Problem Statement: Given a certain amount of resources (e.g., CPU, MEM, etc.) in terms of VMs, resource demands of each job, the dependency constraints of the tasks in each job, and resource capacity constraints of VMs, how to allocate

TABLE I: Features for predicting jobs' execution time.

Feature	Description
Job-related features	
Required CPU	Amount of CPU resource required by the job
Required MEM	Amount of MEM resource required by the job
Required storage	Amount of storage resource required by the job
# of tasks	Number of tasks which the job contains
System-related features	
CPU utilization	VM's CPU utilization
MEM utilization	VM's MEM utilization
Storage utilization	VM's storage utilization

the VM resources to the heterogeneous jobs to achieve higher resource utilization while reducing the latency as much as possible?

C. Job Priority Determination

Figure 3 shows the job classification and priority determination. To classify jobs, Fregata extracts different types of features (i.e., job related features and system related features). Then, Fregata predicts if the job is a short job by predicting the execution time of the job based on the selected features shown in Table I (shown in the left part of Figure 3). If the predicted execution time of the job is no longer than the threshold (T_{th}), then the job is a short job; otherwise, the job is a long job. In the job level, Fregata sets two priority levels: high priority and low priority. If the predicted job is a short job, the job is a high priority job; otherwise, the job is a low priority job.

D. Task Priority Determination

Previous research has shown that assigning high priority to tasks with shorter remaining time can help increase the throughput and reduce latency [18]. However, this can cause starvation of long running tasks. To avoid starvation of long running tasks and reduce the waiting time of each task, Previous methods determine task priority based on task remaining time and (or) waiting time. However, they neglect dependency in the priority determination, which is an important factor to consider to reduce the latency. We consider the priority of task T_{ij} as a function of its remaining time t_{ij}^{rem} and its waiting time t_{ij}^{w} based on the dependency relations among T_{ij} and its children (i.e., the tasks depending on T_{ij}). We follow the method in the work [19] to determine task priority. Table II shows the main notations used in this paper.

E. Task Dependency

We use a directed acyclic graph (DAG) G to model task dependency, and we present a partition based method to find the dependency relations among tasks. Specifically, we partition the nodes into groups based on the dependency relations among them. First, we arbitrarily choose two different nodes: *source* (parentless node of G) and *sink* (childless node of G), then we put all the nodes on the directed path in a group if there exists a directed path from the source to the sink. By following this way, we can partition the tasks that have dependency relation into the same group. The specific steps for partitioning tasks based on the dependency relations among them are illustrated in Algorithm 1.

Algorithm 1: Task_Partition()

```
Input: A set of jobs consisting of tasks represented by DAGs
Output: The dependency relations among tasks for a given job
1 for $v_i$ in $V$ do
2     if $v_i$ in $V_{source}$ then
3         for $v_j$ in $V$ do
4             if $v_j$ in $V_{sink}$ and $v_i \neq v_j$ then
5                 if $\exists$ a direct path form $v_i$ to $v_j$ then
6                     Put all nodes on the path in a group
7                 else
8                     Put $v_i$, $v_j$ into two different groups
```

To compute the dependency relation, we use the reachability between any two nodes in G to represent the dependency relation between the two nodes, and we use a matrix to represent the reachability.

TABLE II: Notations

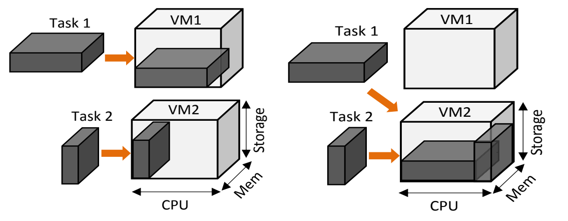
n	Total # of workers	m	# of tasks / job
J	A set of jobs	l	# of resource types
h	# of jobs in J	L_{th}^{s}	Threshold for heavily loaded scheduler
J_i	The ith job in J	$DV(j,i)$	resource deviation of two tasks
T_{ij}	The jth task of J_i	L_{th}^{m}	Threshold for heavily loaded master
t_{ij}^{rem}	T_{ij}'s remaining time	t_{ij}^{w}	T_{ij}'s waiting time

F. Improving Resource Utilization

Fregata presents task packing to improve the resource utilization. For newly arriving tasks, Fregata conducts packing to pack complementary tasks, and then allocates resources to the packed tasks based on their resource demands. The task packing is used to avoid resource fragmentation and achieve high resource utilization. In the following, we first present an example to show the complementary task packing, and then explain its algorithm. Then, we present the resource allocation algorithm that allocates the resources to newly arriving tasks. Finally, we present an example to show this algorithm.

Figure 4 shows an example illustrating how packing strategy decreases the resource fragmentation and increases resource utilization. In Figure 4(a), task 1 (CPU intensive) and task 2 (storage intensive) are assigned to VM1 and VM2, respectively, which increases servers' resource fragmentation. However, in Figure 4(b), task 1 and task 2 are packed first and then assigned to VM2, which releases VM1, and thus decreases the resource fragmentation of servers and increases the resource utilization.

Each task has a dominant resource, defined as the one that requires the most amount of resource. Fregata first packs the tasks with complementary dominant resources such that the summation of the deviation of the two tasks' resource demands on each resource type is the largest. Given a list of tasks, Fregata fetches each task T_i, and tries to find its complementary task from the list to pack with T_i. Note that it is possible that task T_i's complementary task cannot be found from the list. In this case, the task T_i solely constitutes an entity to be allocated with resources in a server. To find T_i's complementary task, Fregata calculates its deviation with



(a) Without packing (b) With packing

Fig. 4: Allocate the resource of VMs to the tasks W/o and W/ packing strategy.

every other task T_j if T_j has different dominant resource from T_i. The deviation is calculated by Equ. (1)

$$DV(j,i) = \sum_{k=1}^{l}((d_{jk} - \frac{d_{jk}+d_{ik}}{2})^2 + (d_{ik} - \frac{d_{jk}+d_{ik}}{2})^2), \quad (1)$$

where d_{ik} is T_i's resource demand on resource type k. Finally, the task with the highest deviation value is the complementary task of T_i. This method can also be applied to tasking packing for three tasks.

G. Resource Allocation

After task packing, Fregata needs to assign each task entity (packed tasks or a task) to a server with the resources. Among the servers with the resources that can satisfy the resource demand of the task entity, Fregata chooses the server that has the least remaining resources (called most matched server) in order to more fully utilize resources. To find the most matched VM, we introduce a concept called unused resource volume. Suppose the vector of the maximum capacity of each resource type among all VMs is $\mathbf{C}' =< C'_1, C'_2, \dots, C'_l >$. Assume that the amount of unused resource of VM j is $\hat{R}_j = (\hat{r}_{j1}, ..., \hat{r}_{jl})$. Then, the unused resource volume of VM j is calculated by

$$volume_j = \sum_{k=1}^{l} \hat{r}_{jk}/C'_k, \quad (2)$$

The VM satisfying the resource demand and has the smallest unused resource volume is the most matched VM.

Figure 5 shows an example illustrating the process of task packing and how Fregata allocates the unused resource to a task entity. For VMs, the numerical values indicate the capacities of different resource types. For tasks, the numerical values indicate the resource demands of tasks. Task 3, task 4, task 5 and task 6 are new arriving tasks. The dominant resource of tasks 3 and 6 is CPU, and the dominant resource of tasks 4 and 5 is storage. Fregata first conducts task packing. The resource demand deviation of task 3 and task 4 is 25, and that of task 3 and task 5 is 16. Since $25 > 16$, task 3 and task 4 are packed together. Similarly, task 5 and task 6 are packed together. We denote the task entities as (task 3, task 4) and (task 5 and task 6). The maximum CPU, MEM and storage of all VMs among both servers are $\mathbf{C}' =< 25, 2, 30 >$. If the amount of unused resource of VMs 1-4 are as follows: $< 5, 0, 20 >$, $< 10, 1, 10 >$, $< 20, 2, 30 >$ and $< 10, 1, 8.5 >$, respectively, based on Equ. (2), their unused resource volumes are 0.867, 1.233, 2.8, 1.183, respectively. To allocate resources

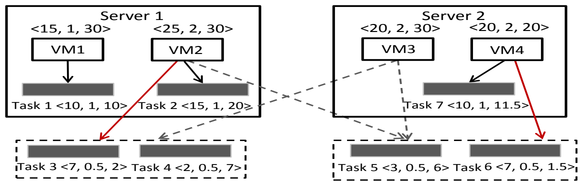


Fig. 5: Allocate unused resource to (packed) tasks with low resource wastage.

to entity (task 3, task 4), Fregata first checks if the VMs' unused resources can satisfy the demands on each resource type of the entity. Then, Fregata chooses the VM that has the smallest unused resource volume to allocate it to the task entity. In this example, VM1 and VM4 cannot satisfy its resource requirements of the packed task (tasks 3, 4). By comparing the unused resource volumes of VM2 and VM3, because $1.233 < 2.8$, then Fregata chooses VM2 rather than VM3 and allocates its unused resource to the packed task (task 3 and task 4). Similarly, the unused resource of VM1 cannot satisfy the resource requirements of the packed task (task 5, task 6). By comparing the unused resource volumes of VM2, VM3 and VM4, because $1.183 < 1.233 < 2.8$, then Fregata chooses VM4 and allocates its unused resource to the packed task (task 5 and task 6). The above process of allocating unused resource to tasks also applies to the single task case.

III. Design

In this section, we present the design of Fregata.

A. Job Submission and Job Assignment

Fregata first uses the machine learning algorithm Support Vector Machine (SVM) to classify the jobs into long jobs and short jobs by predicting the execution time of jobs based on the extracted features. If the predicted execution time of the job is no longer than the threshold (T_{th}), then the job is short job; otherwise, the job is long job. Fregata trains the SVM classifier using the sigmoid kernel function. Fregata repeatedly divides the dataset into training and testing sets using 10-fold cross-validation with percentage split and performs classification.

In Fregata, when users submit their jobs, the jobs are delivered to the schedulers near them. If the scheduler is heavily loaded, the job will be delivered to the lightly loaded neighbor of the heavily loaded scheduler to achieve load balance. Each scheduler has a unique key, and each master has the resource information of the workers associated with master. The workers periodically report their resource information to the master, and the master periodically updates a table which records the resource information of each worker associated with the master.

B. Task Scheduling

Figure 6 shows the architecture of the Fregata. In Fregata, schedulers are scattered in a distributed system. When a user submits a job to the cloud system, the system will deliver the job to the scheduler that is not heavily loaded and has

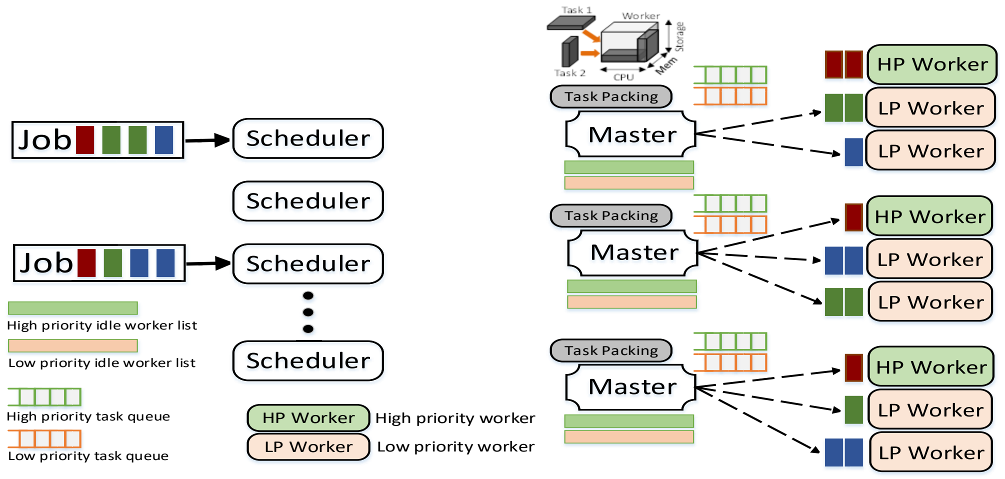


Fig. 6: Architecture of Fregata.

the smallest geographic distance to the user. The scheduler first splits the job into tasks, then it distributes the tasks to m randomly selected masters that are not heavily loaded (Algorithm 2 shows the processing of jobs). The masters pack tasks that have complementary resource requirements based on the task packing strategy in Section II-F. Each master maintains two task queues, where the high priority and low priority queues store tasks belonging to short and long jobs, respectively. Next, masters assign task entity to workers based on the strategy in Section II-G, and the workers calculate the priority of tasks assigned to the workers and run tasks.

Algorithm 2: Job_Processing()

```
Input: A submitted job
1 A Job is submitted
2 sort(s[]) // Sort schedulers by distances between
      schedulers and the user submitting the job
3 for $i \leftarrow 1$ to $Len(s)$ do
4     if $s[i]$ is lightly loaded then
5         Assign the job to $s[i]$
6         $s[i]$ split the job into $m$ tasks
7         $s[i]$ distributes the tasks to $m$ randomly selected masters
            that are not heavily loaded
```

IV. Performance Evaluation

In this section, we present our trace-driven experimental results. We first conducted experiments on a large-scale real cluster Palmetto, which is Clemson University's high-performance computing (HPC) resource [20]. We tested multiple evaluation metrics and compared our method with two other methods. To further evaluate the performance of our method, we conducted experiments on the real-world Amazon EC2 [21]. In the following, we present the experimental results on a real cluster and the experimental results on Amazon EC2, respectively.

A. Trace-Driven Experimental Results

We deployed our testbed in a large-scale cluster Palmetto [20]. We implemented our method and other two methods in our testbed. We compared the results of our method and the other two methods Pigeon [2] and Sparrow [14]. We used up to 1,000 heterogeneous short jobs which have different resource requirements.

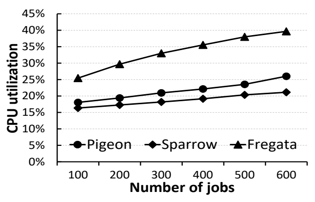
TABLE III: Parameter settings.

Parameter	Meaning	Setting
n	# of servers	30-50
h	# of jobs	100-2500
m	# of tasks of a job	10-2000
L^s_{th}	Threshold for heavily loaded scheduler	0.85
L^m_{th}	Threshold for heavily loaded master	0.85
γ	A certain coefficient $\in (0, 1)$	0.5
ω_1	Weight for task's remaining time	0.5
ω_2	Weight for task's waiting time	0.3
ω_3	Weight for task's allowable waiting time	0.2

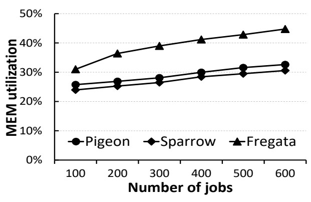
Pigeon [2]. Pigeon is a two-layer, hierarchical scheduler for heterogeneous jobs, which aims to ensure low latency. Pigeon divides workers in a cluster into groups and delegates task scheduling in each group to a group master. Pigeon assigns the tasks of an incoming job to the masters as evenly as possible, and the masters centrally manages all the tasks handled by the corresponding groups. All the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs.

Sparrow [14]. Sparrow is a stateless decentralized scheduler that provides near optimal performance using two key techniques: batch sampling and late binding. Sparrow provides fine-grained task scheduling, which is complementary to the functionality provided by cluster resource managers. Sparrow handles two types of constraints, per-job and per-task constraints. Such constraints are commonly required in data-parallel frameworks.

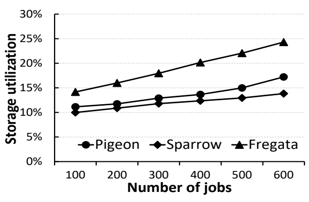
We first deployed our testbed on 50 servers in a real cluster. The servers in the real cluster are from Sun X2200 servers (AMD Opteron 2356 CPU, 16GB memory). We then conducted experiments on 30 instances in the real-world Amazon EC2 and the instances in EC2 are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory). We considered each instance as a server. Each server (instance) was set to have 1GB/s bandwidth and 720GB disk storage



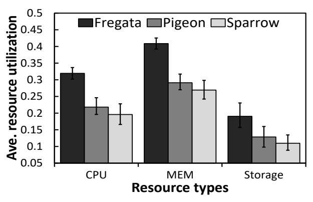
(a) CPU



(b) Memory



(c) Storage



(d) Average

Fig. 7: Utilizations of different resource types vs. number of jobs of different methods on a real cluster.

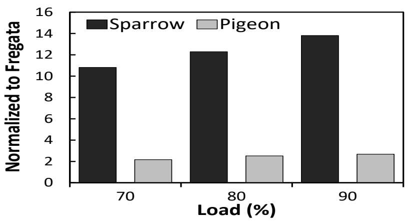


Fig. 8: Job completion times normalized to Fregata on a real cluster.

capacity in both real cluster and EC2 experiments.

In each experiment, we varied the number of jobs from 100 to 600 with step size of 100. The job arrival rate was set to x jobs per minute and x was randomly chosen from [2,5] [22]. The Google Cluster trace [22] records resource usage on a cluster of about 11,000 machines from May 2011 for 29 days. We randomly chose tasks from the jobs in the period between May 1 to May 7. The CPU and memory consumption, and execution time for each task were set based on the Google cluster trace, and the disk and bandwidth consumption for each task was set to 0.02MB [23] and 0.02 MB/s [24], [25], respectively. In the experiment, we created the dependency relationship among tasks based on their starting time and ending time from the trace. When there is no overlap between the execution times of two tasks of a job, we can create a dependency relationship between the two tasks. We constrained the number of levels in a created dependency DAG within five and the number of dependent tasks on a task within fifteen [26]. Table III shows the parameter settings in our experiment unless otherwise specified.

For Fregata, we first use the machine learning algorithm SVM to classify the jobs into long jobs and short jobs by predicting the execution time of jobs based on the extracted features. In the job level, Fregata considers short job as high priority job, and considers long job as low priority job. After classifying jobs, Fregata deliveries the jobs to the scheduler which splits the jobs into tasks. Fregata uses the task packing strategy to pack tasks that have complementary resource requirements, and then allocates the resource to the task entity.

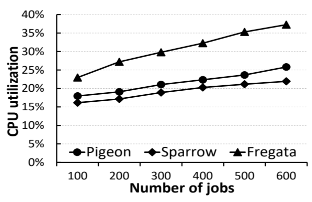
B. Experimental Results on A Real Cluster

Figure 7 shows the relationship between the resource (CPU, Memory and Storage) utilization and the number of jobs on a real cluster. In Figure 7, we see that the resource utilization follows Fregata>Pigeon>Sparrow. The resource utilization in Fregata is higher than that in Pigeon because Fregata leverages complementarity of tasks' demands on different resource types and uses a task packing strategy to reduce the resource fragmentation and improve the resource utilization. The resource utilizations in Pigeon is higher than that in Sparrow because all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs. To verify the performance of resource utilization of different methods, we also measured the average resource utilization of different resource types in different methods. Figure 7(d) shows the average resource utilization of different resource types in different methods. We observe that the average resource utilization follows Fregata>Pigeon>Sparrow due to the same reasons.

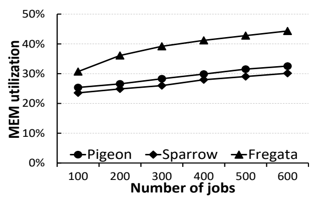
We evaluated the overhead of different methods by measuring the job completion time in each method on the real cluster. We followed the work [2], [4] to measure the job completion time of Pigeon and Sparrow normalized to Fregata under different cluster loads. Figure 8 shows different methods' job completion time normalized to Fregata under different cluster loads. In Figure 8, we see that the job completion time follows Fregata<Pigeon<Sparrow. This is because Fregata splits jobs into tasks by taking into account the constraints (e.g., the dependency among tasks) of tasks , and assigns the tasks that do not depend on each other to different machines so that these tasks can be processed in parallel and the latency can thus be reduced. Also, Fregata leverages task dependency to determine task priority and adequately considers task dependency in scheduling to further reduce latency. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 14 and 5.3 times longer job completion time than Fregata and Pigeon, respectively.

C. Experimental Results on Amazon EC2

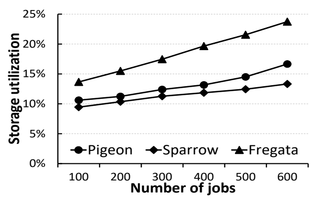
To fully test the performance of our method, we also conducted experiments on the real-world Amazon EC2. Figure 9 shows the relationship between the resource (CPU, Memory and Storage) utilization and the number of jobs on Amazon EC2. In Figure 9, we see that the resource utilization follows Fregata>Pigeon>Sparrow. The resource utilization in Fregata is higher than that in Pigeon because Fregata leverages complementarity of tasks' demands on different resource types and uses a task packing strategy to reduce the resource fragmentation and improve the resource utilization. The resource utilizations in Pigeon is higher than that in Sparrow because all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority



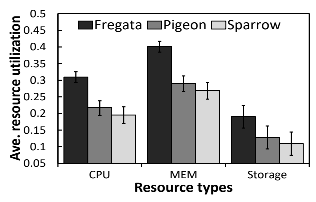
(a) CPU



(b) Memory



(c) Storage



(d) Average

Fig. 9: Utilizations of different resource types vs. number of jobs of different methods on a real cluster.

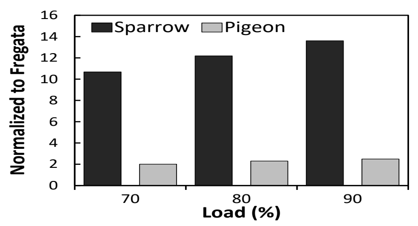


Fig. 10: Job completion times normalized to Fregata on Amazon EC2.

workers are shared by tasks from long jobs. To verify the performance of resource utilization of different methods, we also measured the average resource utilization of different resource types in different methods. Figure 9(d) shows the average resource utilization of different resource types in different methods. We observe that the average resource utilization follows Fregata>Pigeon>Sparrow due to the same reasons.

We also measured the job completion time of Pigeon and Sparrow normalized to Fregata under different cluster loads on Amazon EC2. Figure 10 shows different methods' job completion time normalized to Fregata under different cluster loads. In Figure 10, we see that the job completion time follows Fregata<Pigeon<Sparrow. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 14 and 5.6 longer job completion time than Fregata and Pigeon, respectively. The result in Figure 10 is consistent with that in Figure 8, and the reasons are the same as that explained in Figure 8.

V. Related Work

First, we review previous studies on job (task) scheduling. Then, we describe previous studies on resource utilization related to job (task) scheduling. Finally, we indicate the advantages of our proposed job scheduling Fregata compared to the previous studies.

A. Scheduling

There is a large body of work on scheduling in distributed systems. Many existing studies focused on independent tasks, while many tasks are not independent such as MapReduce tasks. Harchol-Balter [27] analyzed several natural task assignment policies and proposed a new one TAGS (Task Assignment based on Guessing Size). The TAGS algorithms is counterintuitive in many respects, including load unbalancing, non-work conserving, and fairness. The work [27] assumes tasks are independent. Dean and Barroso [28] proposed a scheduling approach using hedged requests where the client sends each request to two worker machines (workers) and cancels remaining outstanding requests when the first result is received. They also described tied requests, where clients send each request to two servers, but the servers communicate directly about the status of the request: when one server begins executing the request, it cancels the counterpart. However, each task must be scheduled independently to target an environment, thus tasks in a job cannot share the information. Scheduling highly parallel jobs that complete in hundreds of milliseconds poses a big challenge for task schedulers. To address this challenge, Ousterhout *et al.* [14] demonstrated that a decentralized, randomized sampling approach provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. However, it neglects the dependency between tasks and thus cannot reduce the latency to the minimum by running tasks that are independent of each other in parallel. Wang *et al.* [2] proposed a two-layer, hierarchical scheduler for heterogeneous jobs, which aims to ensure low latency. Pigeon divides workers in a cluster into groups and delegates task scheduling in each group to a group master. Pigeon assigns the tasks of an incoming job to the masters as evenly as possible, and the masters centrally manage all the tasks handled by the corresponding groups. However, Pigeon neglects the complementary resource requirements of tasks, and it cannot fully increase the resource utilization. Also, it does not leverage task dependency information to determine task priority for reducing the latency.

B. Resource Utilization

There is a large body of scheduling-related work aiming at improving resource utilization. However, many existing studies ignore the resource constraints, and thus the proposed strategies in these studies are not realistic. The work [29] demonstrates that over 50% of the jobs at Google have restrict constraints about the machines that they can run on. Fair schedulers such as Quincy [9] and the Hadoop Fair Scheduler [11] take into account data locality, but these schedulers treat locality as a preference rather than a resource constraint (i.e., hard constraint), and can assign tasks non-locally if a suitable machine is not available. Thus these schedulers neglect jobs' resource constraints existing in practical scenarios. Max-min fairness has been widely studied in various areas such as networks, operating systems, and queuing systems [30]–[34]. However, previous studies have a common unrealistic assumption, that is, the resources required by tasks are identical. Although Dominant Resource

Fairness (DRF) [35] extends max-min fairness to multiple resource types (i.e., CPU, Mem), it also assumes that the resources of a given type such as CPU are identical.

Unlike the existing approaches, our proposed Fregata judiciously utilizes task dependency information for determining task priority by prioritizing tasks that will subsequently enable the execution of more dependent tasks, which helps reduce the latency. Also, Fregata considers the complementary of tasks' requirements on different resource types, and packs complementary tasks, and then allocates resources to the packed tasks, which helps improve the resource utilization.

VI. Conclusions

This paper presents Fregata, a low latency and resource-efficient scheduling for heterogeneous jobs with constraints in clouds. Fregata judiciously utilizes task dependency information for determining task priority by prioritizing tasks that will subsequently enable the execution of more dependent tasks, which helps reduce the latency. Also, Fregata assigns the tasks that do not depend on each other to different workers so that these tasks can be processed in parallel and the latency can thus be reduced. In addition, Fregata considers the complementary of tasks' requirements on different resource types, and packs complementary tasks, and then allocates resources to the packed tasks based on their resource demands, which helps reduce the resource wastage (caused by resource fragmentation) and improve the resource utilization. We compare our method with the existing methods under different scenarios using a real cluster and Amazon EC2 cloud service, and demonstrate Fregata outperforms the exiting methods under both the real cluster and Amazon EC2 cloud service. In the future, we will examine the relationship among the accuracy of job classification using SVM, the resource utilization and the latency for further optimizing the performance of Fregata, and we will compare Fregata with the state-of-the-art hybrid scheduler. Also, we will consider data locality, fairness, cross-job dependency, and the scenario that new tasks are dynamically added which extends the task-dependency graph. In addition, we will consider fault tolerance in designing a low-latency and resource-efficient scheduling system so that the system can handle node failures/crashes or straggler.

Acknowledgment

This research was supported by the Faculty Research Awards Program at Florida A&M University.

References

[1] H. Tian, Y. Zheng, and W. Wang. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In *SoCC*, 2019.

[2] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, and H. Jiang. Pigeon: an effective distributed, hierarchical datacenter job scheduler. In *Proc. of SoCC*, 2019.

[3] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proc. of SoCC*, 2018.

[4] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *SoCC*, 2016.

[5] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proc. of EuroSys*, 2016.

[6] J. Liu and L. Cheng. Swifts: A dependency-aware and resource efficient scheduling for high throughput in clouds. In *Proc. of INFOCOM WKSHPS*, 2021.

[7] J. Liu, H. Shen, and H. S. Narman. CCRP: Customized cooperative resource provisioning for high resource utilization in clouds. In *Proc. of IEEE Big Data*, Washington D.C., 2016.

[8] A. Fergusin, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. of SOSP*, 2009.

[10] A. Tumanov, T. Zhu, J. Park, M. Kozuch, Mor Harchol-Balter, and G. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. of EuroSys*, 2016.

[11] 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 EuroSys*, 2010.

[12] I. Gog, M. Schwarzkopf, A. Gleave, R. Watson, and S. Hand. Firmanent: Fast, centralized cluster scheduling at scale. In *Proc. of OSDI*, 2016.

[13] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SoCC*, Santa Clara, 2013.

[14] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of SOSP*, 2013.

[15] M. Khelghatdoust and V. Gramolim. Peacock: Probe-based scheduling of jobs by rotating between elastic queues. In *Proc. of Euro-Par*, 2018.

[16] K. Karanasos, S. Rao, C. Douglas, K. Chaliparambil, G. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of ATC*, 2015.

[17] P. Delgado, F. Dinu, A. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of ATC*, 2015.

[18] L. E. Schrage and L. W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4), 1996.

[19] J. Liu, H. Shen, A. Sarker, and W. Chung. Dependency in scheduling and preemption for high throughput in data-parallel clusters. In *Proc. of IEEE CLUSTER*, Belfast, United Kingdom, 2018.

[20] Palmetto cluster. http://citi.clemson.edu/palmetto/ [accessed in December 2021].

[21] Amazon ec2. http://aws.amazon.com/ec2 [accessed in December 2021].

[22] Google trace. https://code.google.com/p/googleclusterd-ata/ [accessed in July 2017].

[23] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of FAST*, 2012.

[24] A. L. Shimpi. The SSD anthology: Understanding SSDs and new drives from OCZ, 2014.

[25] J. Liu, H. Shen, and L. Chen. CORP: Cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. In *Proc. of IEEE CLUSTER*, 2016.

[26] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of OSDI*, 2016.

[27] M. Harchol-Balter. Task assignment with unknown duration. *Journal of the ACM*, 49(2):260–288, March 2002.

[28] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[29] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proc. of SoCC*, 2011.

[30] B. Avi-Itzhak and H. Levy. On measuring fairness in queues. *Advanced in Applied Probability*, 36:919–936, 2004.

[31] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition edition, 1992.

[32] D. Raz, H. Levy, and B. Avi-Itzhak. A resource-allocation queueing fairness measure. In *SIGMETRICS Perform. Eval. Rev.*, volume 32, pages 130–141, June 2004.

[33] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proc. of OSDI*, 1994.

[34] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an m/gi/1. In *Proc. of SIGMETRICS*, 2003.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, I. Stoica, and S. Shenker. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. of NSDI*, 2011.