Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler

Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun*, Jiao Rao, Hao Che and Hong Jiang
The University of Texas at Arlington **Guangdong Phamaceutical University*

September 3, 2019

Abstract

In today's datacenters, job heterogeneity makes it difficult for schedulers to simultaneously meet latency requirements and maintain high resource utilization. The state-of-the-art datacenter schedulers, including centralized, distributed, and hybrid schedulers, fail to ensure low latency for short jobs in large-scale and highly loaded systems. The key issues are the scalability in centralized schedulers, ineffective and inefficient probing and resource sharing in both distributed and hybrid schedulers.

In this paper, we propose Pigeon, a distributed, hierarchical job scheduler based on a two-layer design. Pigeon divides workers into groups, each managed by a separate master. In Pigeon, upon a job arrival, a distributed scheduler directly distribute tasks evenly among masters with minimum job processing overhead, hence, preserving highest possible scalability. Meanwhile, each master manages and distributes all the received tasks centrally, oblivious of the job context, allowing for full sharing of the worker pool at the group level to maximize multiplexing gain. To minimize the chance of head-of-line blocking for short jobs and avoid starvation for long jobs, two weighted fair queues are employed in in each master to accommodate tasks from short and long jobs, separately, and a small portion of the workers are reserved for short jobs. Evaluation via theoretical analysis, trace-driven simulations, and a prototype implementation shows that Pigeon significantly outperforms Sparrow, a representative distributed scheduler, and Eagle, a hybrid scheduler.

1 Introduction

Workload heterogeneity has been a long-standing challenge in datacenter scheduling. Jobs that differ in execution time and fanout degree have distinct requirements for scheduling. Short jobs have stringent latency requirements and are sensitive to scheduling delays; long jobs, which usually have a large fanout and high resource demands, require high-quality scheduling, e.g., improving load balance, but can tolerate some scheduling delays. While short jobs are usually user-facing applications [3, 16] and important to user-perceived quality-of-service, long jobs help improve datacenter resource utilization. Therefore, it is common practice to collocate short and long jobs in datacenter management, but meeting the diverse needs of heterogeneous jobs remains a critical challenge.

Early datacenter job schedulers, e.g., Jockey [11], Quincy [15], Tetrished [26], Delay Scheduling [28] are centralized by design. Centralized schedulers rely on a global view of resource availability to make scheduling decisions. As systems scale, handling a large number of jobs and collecting runtime status from a large number of nodes inevitably become a bottleneck and incur a significant scheduling delay for each job. This is particularly problematic for short jobs with tight deadlines.

To address the scalability issue, recent research, such as Sparrow [19] and Peacock [18], employs multiple schedulers to dispatch tasks in an independent and distributed manner. Without requiring a global view of resources, distributed schedulers probe randomly selected nodes (usually twice as many as the number of tasks to be dispatched) and dispatch tasks onto the least loaded nodes. The probe based technique has been proved to greatly improve task queuing time compared to random placement [19]. However, each scheduler still needs to maintain a fairly large amount of probe related states and incurs non-negligible probe processing overheads.

Besides the above issues, the collocation of heterogeneous workloads presents unique challenges to the centralized and distributed schedulers. First, heterogeneous workloads require an effective mechanism to prioritize short jobs over long jobs. Distributed schedulers lack coordination among one another, thereby unable to enforce global service differentiation among jobs. While centralized schedulers can employ priority queues to differentiate task scheduling for different types of jobs, they are usually work conserving – low priority, long jobs can utilize the entire cluster to avoid wasting cluster resources.

However, by doing so, a burst of long jobs can inflict the so called head-of-line blocking to short jobs that arrive immediately after the burst. Even in the presence of centralized priority queues, tasks from short jobs need to wait for the tasks of long jobs that have already been dispatched onto workers. Recent work BigC [4] and Karios [8] propose to suspend long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks. Second, high resource utilization in datacenters that embrace workload consolidation makes randomized load balancing less effective. For heterogeneous workloads that contain tasks of various sizes, it is difficult to identify less loaded nodes. It has also been reported that randomized load balancing is inefficient and requires multiple rounds of probing to locate idle or less busy nodes if most nodes are highly loaded [25].

Hybrid approaches, such as Mercury [17], Hawk [10] and Eagle [9], combine centralized and distributed schedulers, with former handling long jobs and the latter short jobs. However, long and short jobs are scheduled independently. This makes it difficult to mitigate the negative impact of long jobs on the performance of short jobs. For example, Eagle [9] employs two techniques to entirely eliminate the head-of-line blocking, i.e., multiple rounds of probing for short-job task placement and a reserved worker pool for short jobs. However, as the cluster load becomes high, most of the short jobs are driven by long jobs to the reserved pool [8], resulting in rapid performance deterioration for short jobs. Our simulations based on the Yahoo trace [6] show that the performance of short jobs drastically degrades, by as many as 70 times at high load compared with the non-resource-constrained case (see Section 4 for details).

In this paper, we demonstrate that a hierarchical scheduler that employs a divide-and-conquer approach in task scheduling can effectively overcome the shortcomings of centralized, distributed and hybrid schedulers, and ensure low latency for short jobs while maintaining high resource utilization without significant sacrificing the performance of long jobs. To this end, we propose *Pigeon*, a two-layer, hierarchical scheduler for heterogeneous jobs. Pigeon divides workers into groups and delegates task scheduling in each group to a group master. Upon job submission, Pigeon assigns the tasks of an incoming job to the masters as evenly as possible. The dispatching of tasks onto masters is intended to be simple and does not consider the type of tasks. The master in each group implements more sophisticated scheduling by maintaining two weighted fair queues, one for tasks from short jobs and the other for tasks from long jobs, respectively, and partitioning workers in each group into high and low priority workers. Tasks of short jobs can run on any workers while tasks of long jobs can only run on low priority workers. Tasks are only dispatched when there are idle workers from a group and are otherwise queued at a respective priority queue according to their types.

Although hierarchical construct has been widely used in distributed systems, such as two-level hierarchical routing structure for the Internet and clustering for Peer-to-Peer systems, to the best of our knowledge, Pigeon is the first hierarchical solution purposely designed and used for job scheduling in datacenters. Pigeon's two-layer design is specially useful for heterogeneous jobs. First, it effectively mitigates head-of-line blocking of short jobs. The simple job-oblivious task dispatching among masters prevents a burst of tasks from monopolizing all workers and provides a certain level of isolation between jobs. Unlike in a centralized scheduler, where tasks of the same type (i.e., short jobs) can only be served in FIFO order, tasks of different jobs in Pigeon are evenly distributed among masters, allowing tasks that arrive late to start to execute even before some tasks of an earlier job start to execute (see Section 4.1 for details). Second, the two-layer design preserves good scalability of distributed schedulers but avoids the pitfalls of randomized load balancing. The size- and type-oblivious task dispatching among masters provide sufficient randomness for effective load balancing without global knowledge and the weighted fair queuing based scheduling within a group is deterministic, ensuring that idle workers are rapidly located to serve latency-sensitive jobs without starving the long jobs.

We perform an evaluation of Pigeon through theoretical analysis, simulation, and a prototype implementation on the Amazon EC2 cloud. Analysis results show that Pigeon can greatly increase the job-zero-queuing probability compared to Sparrow, a representative distributed scheduler, for workloads that only contain short jobs. Trace-driven simulations based on the Yahoo, Cloudera and Google traces demonstrate that Pigeon outperforms Eagle, a state-of-the-art hybrid scheduler, on short job performance by as many as tens of times in a highly loaded cluster. Experimental results on the Amazon EC2 also confirm the effectiveness of Pigeon.

2 Pigeon Scheduler

This section presents Pigeon. We first give an overview of Pigeon and introduce its task placement scheme, and then discuss how it handles tasks at the master level.

2.1 System model

We consider a datacenter cluster composed of a large number of workers, each of which can be an independent processing unit, such as a CPU core. The workers can run in parallel to execute different tasks. A key idea in Pigeon

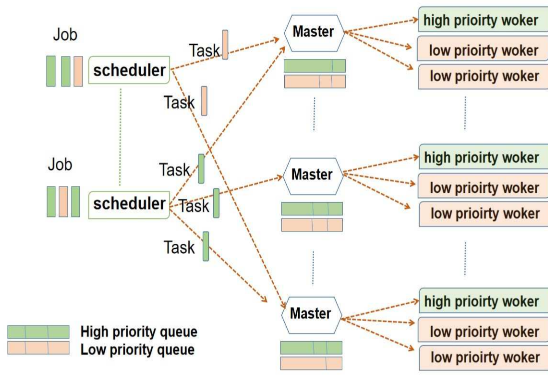


Figure 1: Overview of Pigeon.

is to divide workers into groups. Each group is managed by a master which centrally controls all the tasks handled by the group and places tasks among the workers in the group. Distributed job schedulers directly distribute the tasks belonging to a job to the masters. After a master receives a task, it either directly sends the task to an idle worker to be processed immediately or puts it in the corresponding task queue if there is no idle worker in the group at the time. Figure 1 gives a system overview of Pigeon. The system is composed of multiple distributed job schedulers, masters, and workers. All job schedulers work independently and do not exchange any task placement information among themselves.

A master works at the task level and is mostly job oblivious except for its awareness of whether a task is a low or high priority one, based on whether the task is from a short or a long job. It maintains two weighted fair task queues, where the high priority and low priority queues store tasks belonging to short and long jobs, respectively. The classification of a job as a short or long job is handled by schedulers, based on the type of application the job belongs to. For example, user-facing applications, such as web searching and social networking, that generally have short task execution times and require stringent tail latency guarantee, can be classified as short jobs. On the other hand, background batch applications, such as data backup, that usually have long task execution times and call for loose mean response time assurance, can be classified as long jobs. In Pigeon, a small number of workers in the group, called high priority workers, are reserved exclusively for serving high priority tasks. The other workers, called low priority workers, can serve both low and high priority tasks. Since 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, Pigeon can greatly improve resource efficiency, achieving high multiplexing gain, compared with the existing job schedulers that distribute tasks directly to individual workers.

A master can run in a worker who needs to be be allocated enough computation resource to effectively handle group status report and task placement functions. As we shall discuss in more detail later, in Pigeon a master needs to handle about one incoming task per second on average, which is modest from computation resource demand point of view.

2.2 Task Scheduling

Assume that a system has N_s schedulers and N_g groups (i.e., N_g masters). Each group has N_w workers in it. For a job with F tasks (i.e., fanout degree, F), the scheduler that handles the job will distribute the tasks as follows. It sends $S = [F/N_g]$ task(s) to each master (here $[x]$ represents the floor of x, i.e., the integer part of x) and the remaining $r = F\%N_g$ to r randomly selected masters. Since the number of workers in each group is much larger than one (i.e., a range of 50 to 100), according to the law of large numbers, the workloads distributed to different groups are expected to be much more balanced than those distributed directly to individual workers. This helps synchronize the task processing for tasks belonging to the same job and hence, reduce the job completion time, with respect to the existing job scheduling solutions.

Two task queues of different scheduling priorities are set in each master to store the corresponding classes of tasks[1], i.e., tasks belonging to short and long jobs. More specifically, the two queues are scheduled based on weighted fair queuing with a single integer weight to ensure that tasks from the high priority queue are served with higher priority than those from the low priority queue, without starving the low priority tasks. The queue scheduler ensures that out of every W tasks to be served, at least one comes from the low priority task queue if it is not empty. The queue scheduler degenerates to strict priority queuing, when W is set to infinity. In this case, the low priority tasks can be served only when the high priority task queue is empty.

A master maintains two idle worker lists, i.e., the high and low priority idle worker lists that record all high and low priority workers that are currently idle, respectively. A task sent to a master must include the priority of the task. When a master receives a high priority task, it first checks whether the low priority idle worker list is empty or not. If the list is not empty, an idle worker from the list is removed and assigned to handle the task. Otherwise, the master checks whether the high priority idle worker list is empty or not. If it is not empty, a worker is removed from the list and assigned to handle the task. If both idle worker

[1]Pigeon can be easily extended to support more than two job classes by allocating as many priority queues as the number of job classes with weighted fair queuing.

lists are empty, the high priority task is put into the high priority task queue. When a master gets a low priority task, it only checks the low priority idle worker list. If the list is not empty, a worker is removed from the list to serve the task. Otherwise, the task is put into the low priority task queue. Whenever a worker is selected to handle a task, the master sends the task to the worker, together with the scheduler identifier (ID) for the scheduler from which the task is received. If a master receives multiple tasks from a job at a time, it handles these tasks one by one consecutively following the same procedure.

We note that both reserving a given portion of workers in a group for high priority tasks and setting W to be a finite integer help to avoid head-of-line-blocking of short jobs and starvation of long jobs, respectively. The exact values of these two parameters must be properly selected in practice. For all our real-world-trace-driven case studies (see Section 4), we found that no more than 10% of workers need to be reserved to achieve high short job performance, lower than that of Eagle, a state-of-the-art hybrid scheduler. In the meantime, W can be simply set to infinity to achieve the highest short job performance without significantly impacting the long job performance. This is because the trace statistics show that the short job execution time is less than 20% of the overall job execution time and hence, long jobs have little chance to be starved by short jobs.

When a worker completes a task, it sends the reports/results directly to the corresponding scheduler and meanwhile, sends an idle notification message to its master. This may further trigger a task in one of the two queues to be sent to the worker or the worker to be added to the high priority worker list if it is reserved for high priority jobs, otherwise, to the low priority worker list.

3 Performance Modeling and Analysis

To gain insights on the Pigeon performance, in this section, we conduct simple performance modeling and analyses for Pigeon, compared with the analysis of a performance model for Sparrow [19]. To be mathematically tractable, we consider only one class of jobs. Hence, only one task queue is used in each master. In this case, all the workers serve tasks from all jobs. We focus on short jobs, which are usually more latency sensitive and whose fanout degrees are smaller than long jobs. We assume that the job fanout degrees are no larger than the number of groups.

Consider a cluster with N_g groups and each group with N_w workers, with a total of $N_c = N_g N_w$ workers in the cluster. Assume that jobs arrive following a Poisson arrival process with average arrival rate λ. All the jobs have fanout degree, F, where $F \leq N_g$, and the task execution time follows an exponential distribution with average execution time, T_e.

With the above model, each master can then be approximately modeled as running a single M/M/N_w task queue [7] with average task arrival rate $\lambda_t = \lambda F / N_g$. The worker utilization is $\rho = \lambda_t T_e / N_w$. Given that $F \leq N_g$, the probability, $P_{task}(0)$, that a task experiences zero queuing time in a group is then given as follows [7],

$$P_{task}(0) = 1 - \frac{1}{1 + (1-\rho)(\frac{N_w!}{(N_w\rho)^{N_w}}) \sum_{k=0}^{N_w-1} \frac{(N_w\rho)^k}{k!}}, \tag{1}$$

and the average queuing time T_q for a task in a master is

$$T_q = \frac{1 - P_{task}(0)}{N_w/T_e - \lambda_t}. \tag{2}$$

In this paper, a job is considered to have zero queuing time if the job completion time (not including the communication time) is equal to its longest task execution time. For example, assume that a job has 2 tasks with execution time 10s and 100s, respectively. If the job completion time is 100s, it experiences no queuing delay, even though its task with 10s execution time may have queued for some time, e.g., 50s.

Now we first consider the case that all the tasks in a job have the same execution time. Then the job-zero-queuing probability in Pigeon, $P_{job}^{PI}(0)$, can be written as,

$$P_{job}^{PI}(0) = (P_{task}(0))^F. \tag{3}$$

In this case, the job-zero-queuing probability for Sparrow, $P_{job}^{SP}(0)$, using $2F$ probes per job, is derived in the original paper on Sparrow [19], as follows,

$$P_{job}^{SP}(0) = \sum_{i=F}^{2F} (1-\rho)^i \rho^{2F-i} C(2F, i), \tag{4}$$

where $C(2F, i)$ is the combination function.

Figure 2 depicts the analytical job-zero-queuing probability for Sparrow (i.e., Eqn.(4)) and Pigeon (i.e., Eqn.(3)) for two different group sizes, i.e., N_w=100 and 200 and two job fanout degrees, i.e., $F = 50$ and 100. As one can see, the job-zero-queuing probability for Sparrow starts to drop at load 0.4 and quickly drops to near zero at load 0.6, whereas for Pigeon, similar drops occur in a much higher load region, i.e., 0.6 to 0.8. It means that Pigeon can work at 20 - 40% higher load than Sparrow, while achieving similar job-zero-queuing performance as Sparrow, demonstrating the effectiveness of Pigeon for job scheduling, compared with Sparrow.

We also note that for Pigeon, when N_w increases from 100 to 200, the job-zero-queuing probability starts to drop at load 0.7, 0.1 higher than the former case. But it quickly

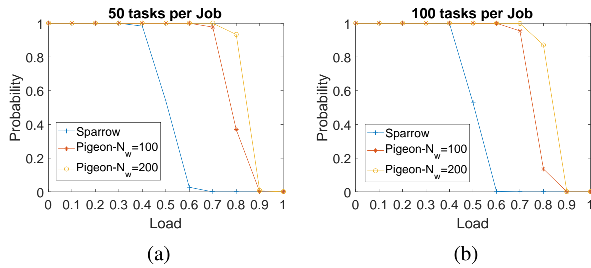


Figure 2: Job-zero-queuing probabilities for Sparrow and Pigeon with two different group sizes (N_w=100 and 200). All tasks in a job have the same execution time. (a) Job fanout F=50; (b) F=100.

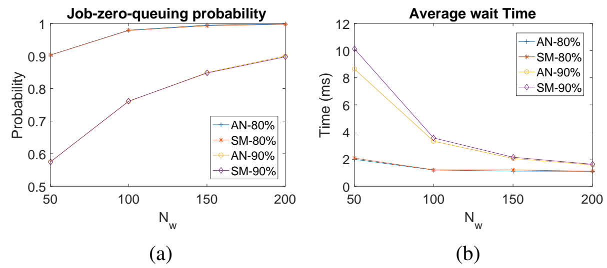


Figure 3: Analysis (denoted as AN) vs Simulation (denoted as SM) at different group sizes. (a) Job-zero-queuing Probabilities (b) Average job wait time.

approaches 0 as the load approaches 0.9, similar to the former case. This suggests that a larger group can improve performance in the medium load region (0.7 to 0.8), but not much in high load region (>0.9).

The above analyses assume that each task in a job has the same execution time. However, real trace analyses indicate that the task execution time can vary significantly from one task to another for a given job. To capture the performance impact of such variability, we consider the case where the task execution time for a task in a given job follows an exponential distribution.

We first calculate the average job queuing time, T_{job}. Since the job queuing time is defined as the queuing time of the slowest task of the job, we need to find the queuing time for the slowest task of the job. To this end, we observe from Figures 3(b) and 4(b) that the average queuing time in Pigeon is much shorter than the average task execution time even at a high load (e.g., 90%). This suggests that whichever task has the largest execution time is likely to be the slowest one, regardless of its queuing time. This implies that the average queuing time for the slowest task can be simply approximated as the average queuing time for all tasks, i.e., $T_{job} \approx T_q$.

Now we calculate the job-zero-queuing probability. Consider two independent exponential distribution random variables (t_1 and t_2) with average value T_e, the joint probability density function $f(t_1, t_2) = \frac{1}{T_e^2} e^{-(t_1+t_2)/T_e}$. Then the probability of $t_1 - t_2 > T_q$ under condition $t_1 > t_2$ [23] is

$$P(t_1 - t_2 > T_q | t_1 > t_2) = e^{-T_q/T_e}. \quad (5)$$

Let $A1$ and $A2$ be the tasks with the longest (t_1) and second longest (t_2) execution times in a job, respectively. Now the job-zero-queuing probability $P_{job}^{dt}(0)$ for a job with different task execution times can be approximately expressed as the probability of $A1$ with zero-queuing time (i.e., $P_{task}(0)$) while $t_1 - t_2 > T_q$, i.e., the execution time difference between the longest and the second longest task execution time is greater than the average task queuing time T_q, namely,

$$P_{job}^{dt}(0) \approx P_{task}(0) e^{-T_q/T_e}. \quad (6)$$

We verify the analytical approximations for T_{job} and P_{job}^{dt} by simulation. Assume that N_c=30,000, $F = 100$, and T_e=100 ms. Each task execution time follows an exponential distribution. The communication time is set at 0.5 ms between any two nodes. We note that with communication delay, the average job waiting time T_w is no longer equal to the average queuing time, but rather the average queuing time plus the communication time.

We study the Pigeon performance by changing N_w from 50 to 200. Figure 3 depicts the job-zero-queuing probability and the average job waiting time at two different high loads (i.e., 80% and 90%). We note that the simulation results (denoted as SM) closely match the analytical ones (denoted as AN), e.g., less than 1% for the job-zero-queuing probability for all N_w's tested. The largest difference is about 12% for the average waiting time at N_w=50 and the load of 90%. In this case, the simulated waiting time (also queuing time) is longer than the analytical one because the analytical results only consider the waiting time for the task with the longest execution time. As the job-zero-queuing probability is low (below 60%), the contribution of other tasks may not be neglected, resulting in larger errors.

The results verify that Eqns. (2) and (6) can be used to estimate the performance of Pigeon for handling jobs with fanout degrees less than the number of groups. The results indicate that the job-zero-queuing probability increases and the average waiting time decreases as the group size increases. It means that a larger group can provide better performance, particularly from 50 to 100. The performance improves slower as the group size increases from 100 to 200, particularly for the average waiting time. Further increasing the group size is expected to offer marginal performance gain. This result provides some insight on how to set the right group size when a cluster handles jobs

Trace	F_{max}	F_{min}	F_{avg}	T_e^{max} (s)	T_e^{min} (s)	T_e^{avg} (s)
Yahoo	5900	1	39.91	21259.9	1.54E-5	118.78
Cloudera	51834	1	272.93	97941.8	3.89E-5	162.19
Google	49960	1	35.32	774922	1E-6	661.74

Table 1: Trace statistics of job fanout and execution time

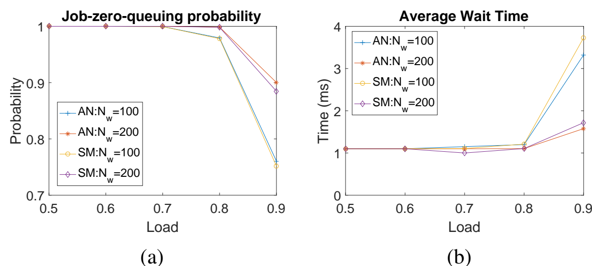


Figure 4: Analysis (denoted as AN) vs Simulation (denoted as SM) results with various cluster loads. (a) Job-zero-queuing probabilities (b) Average job wait time.

with small fanout degree (i.e., the number of tasks in a job is less than the number of groups in the cluster).

Now we study the performance of Pigeon by varying cluster loads. Two cases with N_w set at 100 and 200 are studied. The results are given in Figure 4. Again, the simulation results closely match the analytical ones. The results indicate that the job-zero-queuing probability is close to 1 even at load 80% and reduces to 0.7 at load 90%. This means that most jobs do not need to be queued even at high load, hence offering high probability of meeting the tightest job performance requirements at high load. We also note that the average waiting time is very small (less than 4%) compared to the average task execution time even at very high load, e.g., 90%. These results clearly demonstrate the effectiveness of Pigeon for job scheduling.

The following two sections test the efficiency of Pigeon by large-scale simulation and on a small EC2 cloud cluster, respectively.

4 Simulation Testing

To test the scalability and efficiency, we use simulation to evaluate the performance of Pigeon against Eagle[2] in large clusters, using three real-world traces as input, i.e., Yahoo [6], Cloudera [5], and Google traces [21]. The open source simulation code of Eagle [9] is used and a similar event-driven simulator is developed for Pigeon.

Table 1 provides the statistics of these traces, including the maximum/minimum/average job fanout degrees (denoted as $F_{max}/F_{min}/F_{avg}$) and maximum/minimum/average task execution time (denoted as $T_e^{max}/T_e^{min}/T_e^{avg}$). We see that the job fanout degree ranges from 1 to 51834; the execution time varies from microseconds to over 700K seconds; and the average task execution time ranges from 118.78 seconds to 661.74 seconds. Unlike the modeled workload in the previous section, these statistics indicate that the job size in terms of both fanout degree and task execution time vary significantly from job to job in practice. Such job heterogeneity makes it difficult to meet service requirements for individual applications, e.g., in terms of providing job completion time or throughput guarantee. For example, for a cluster with 10K workers and a long job with fanout degree of 50K, each worker needs to execute 5 tasks for the job on average. The placement of such a job evenly among all the workers in the cluster can take up all the cluster resources at once, causing head-of-line blocking to the upcoming short jobs. As aforementioned, to effectively deal with the job heterogeneity issue, both Pigeon and Eagle [9] reserve a subset of workers to be used by short jobs only, at the group-level and cluster-level, respectively. In what follows, we first discuss the parameter settings, in terms of the short-vs-long job thresholds, the reserved worker pool size, the communication delays, the group size, and the weight value for weighted fair queuing and then performance evaluation.

4.1 Parameter Settings

Short Jobs vs Long Jobs: As mentioned earlier, in practice, a scheduler may rely on whether a job belongs to a user-facing application to classify it as a short job or not. However, due to the lack of the application information for the three traces and to fairly compare against Eagle, for Pigeon, we simply use the same short job cutoff times, defined as the average task execution time of a job, as those used in Eagle, i.e., 90.5811, 272.783 and 1129.532 seconds for the Yahoo, Cloudera

[2]As Eagle outperforms Sparrow and Hawk[10], only Eagle is compared here.

and Google traces, respectively.

Reserved Worker Pool Size: The actual number of workers reserved for tasks of short jobs has significant impact on job completion times for both short and long jobs. The more workers are reserved, the smaller the job completion time for short jobs but the larger the job completion time for long jobs. We study the performance using the three traces by varying the worker reservation ratio (due to page limitation, the results are not presented here). By taking into account of the performance for both short and long jobs, we decide to set the reservation ratios at 2%, 8% and 9% for the Yahoo, Cloudea and Google traces, respectively. For Eagle, against which Pigeon is to be compared in the following section, we set the reservation ratios for the three traces at the same values as those used in [9], i.e., 2%, 9% and 17%, respectively.

Weight Value for Fair Queuing: The weight value W is an important parameter for Pigeon. A smaller (larger) W helps improve the performance of long (short) jobs at the cost of the other. We study the Pigeon performance by varying W from 5 to 100 and compared to that with strict priority queuing (i.e. W is set to infinity) (again, the results are not presented here due to page limitation). We find that the short job performance becomes very sensitive to W at high cluster loads when W gets below 20. For example, while the 99th-percentile short job completion time at $W = 20$ is within 140% of that at $W = \infty$, it increases to more than 300% at $W = 5$, at high cluster loads for all the three traces. Meanwhile, we find that the long job performance is insensitive to W in a wide range, e.g., only 2% difference from $W = 10$ to ∞ at all cluster loads for all the three traces. In other words, no long job starvation occurs even at $W = \infty$ for all the three traces. So for all the simulation studies and the testing on EC2 cloud, we simply set $W = 20$ and ∞, respectively.

Communication Delays: The communication delays are set at 0.5 ms between any two nodes, i.e., a scheduler and a master, a master and a worker, or a worker and a scheduler.

Group Size: Without knowing the exact processing overhead per task scheduling at each master, we have not taken this overhead into account in both performance modeling in the previous section and the simulation in this section. As a result, intuitively, one would expect that the testing results in both previous and this section will be always in favor of larger group size, with the group size equal to the cluster size offering the highest performance (i.e., the case when Pigeon degenerates to a centralized scheduler). While this intuition is confirmed

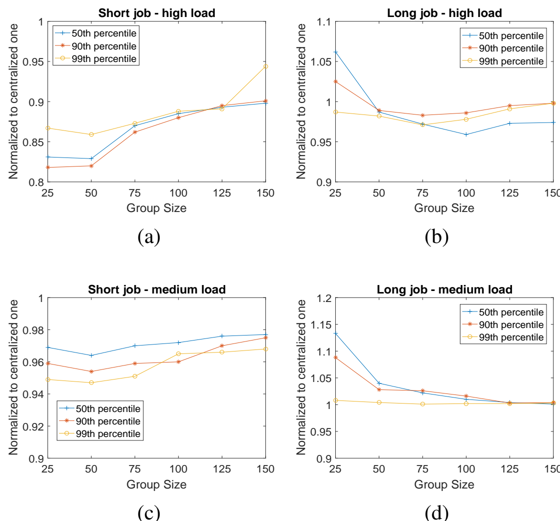


Figure 5: Pigeon performance at different group sizes.

in the previous section based on the results from an ideal model, much to our surprise, it turns out to be false as confirmed by the simulation results in this section. More specifically, we can conclude that *Pigeon with the group size in a finite range actually outperforms its centralized counterpart, even when the centralized scheduler incurs negligible processing overhead.* The implication of this is significant. It means that one can no longer assume that so long as it is free from scalability concerns, centralized scheduling is always the best choice, as it has a global view of the cluster resource availability. In what follows, we first identify the range and the preferred group size, and then provide an explanation of why this seemingly counter-intuitive phenomenon can happen.

We compare the Pigeon performance at different group sizes, using the Cloudera trace as input for the simulation (similar results are obtained for the Yahoo and Google traces and hence are not given here). We consider the cluster size of 12K and 18K, corresponding to high (about 93%) and medium (about 62%) cluster loads, respectively. All other parameters pertaining to Pigeon are the same as those given above. The 50th, 90th and 99th percentiles of the short and long job completion times are used as performance metrics.

From the results depicted in Figure 5 (normalized to the centralized one), we can see that Pigeon performs better than its centralized counterpart for all the three performance metrics for short jobs, particularly at the high load (Figures 5 (a)). At high load, the short job performance gets better as the group size reduces from 150 to 50 and then becomes slightly worse as it further reduces to 25. The largest performance gains for short jobs are about 17%, 18% and 14% for 50th, 90th and 99th percentile

job completion times at group size 50, compared to the centralized one, respectively. Similar results with smaller gains are observed at the medium cluster load (Figure 5 (c)). For long jobs at high load (Figure 5 (b)), the performance is better (worse) than the centralized one when group size is above (below) 50. The three percentiles of job completion times decrease when the group size reduces from 150 to 100, and then increase when the group size further reduces. In the medium load (Figure 5 (d)), the performance for long jobs is worse than that of the centralized one in the entire group size range studied. All the three percentiles of long job completion time decrease as the group size increases.

The above results indicate that Pigeon is not only more scalable but also performs better than its centralized counterpart for handling heterogeneous jobs, particularly at high cluster loads. Based on these observations, which agree with the observation made for the other two traces, and also with reference to the analytical results for jobs at small fanout, we suggest to set group size in a range between 50 and 100 and we set the group size at 100 for all the cases studied in this section.

Explanations for the Counterintuitive Phenomenon: A key observation we make is that this phenomenon may occur when both job fanout degree and task execution time vary in a wide range, which is the case in practice (see Table 1) but not for the model in the previous section (that explains why we did not see this phenomenon there). The best way to see why this is true is by example.

Consider job scheduling for a single type of jobs and a cluster of 4 workers. At time 0, all the workers are idle and job A with 6 tasks arrives, with task execution times of 20, 1, 1, 10, 10 and 10 units. Immediately following it are two other jobs B and C, each having 1 task with execution time of 2 units. We further assume that there is no processing overhead and the communication time can be neglected. Now we compare the performance of a Pigeon scheduler and its centralized counterpart.

First consider a Pigeon scheduler, where 4 workers are divided into 2 groups with 2 workers each. Upon the arrival of jobs A, B, and C, in that order, the first 3 tasks (with execution times 20, 1 and 1) from A are sent to group one and the other 3 tasks (all with execution time 10) to group two. Then the task from job B is sent to group one and the task from job C to group two. In group one, two tasks (with execution times 20 and 1) of job A are served by workers 1 and 2 at time 0, respectively. At time 1, worker 2 completes the task and immediately start serving the remaining task from job A. It finishes the task at time 2 and then serves the task from job B, which completes at time 4. In group two, at time 0, workers 3 and 4 serve 2 tasks from job A and complete the tasks at time 10. Then worker 3 serves the last task from job A and worker 4 serves the task from job C, which finishes at time 12. As job A finishes at time 20, the job completion times for the three jobs are 20, 4 and 12, for a total of 36 units.

Now, consider centralized scheduling. The first 4 tasks (with execution times 20, 1, 1 and 10) from job A are sent to workers 1-4 at time 0, respectively. At time 1, workers 2 and 3 complete their tasks and then serve the other two tasks from job A. At time 10, worker 4 finishes its task and then serves the task from job B, which finishes at time 12. At time 11, workers 2 and 3 complete their tasks, and then worker 2 serves the task from job C which finishes at time 13. Again, job A finishes at time 20. So the job completion times for the three jobs are 20, 12 and 13, respectively, for a total of 45 units, 9 units or 25% more than the Pigeon scheduler!

From the above example, we see that for centralized scheduling, a job with a large fanout degree (i.e., job A) causes head-of-line blocking of the following jobs of the same type, even when their fanout degrees are low (i.e., jobs B and C). In contrast, for Pigeon, the tasks for jobs are distributed to different groups. This enhances the chance for tasks from later jobs to be served before tasks from the earlier jobs due to heterogeneous task execution time distribution. This helps reduce the chance of head-of-line blocking of jobs with small fanout degrees by jobs with large fanout degrees, hence, resulting in better overall performance. While helping more in alleviating head-of-line blocking by dispersing the tasks of a job with a large fan-out degree to more groups, using a smaller group size reduces multiplexing gain. This help explain why Pigeon gives the overall best performance at the group size in a certain range, i.e., 50 to 100.

Master Workload Estimation: Finally, with the parameters given above, we can now estimate the offered task load at a master. Assume that the cluster size, N_c=20,000, and hence, the total number of masters, N_g=200, given the group size, N_w=100. The real trace statistics in Table 1 suggest that the average task execution time is more than 100s (from 118s to 661s, to be exact). It means that a master needs to handle about only 1 task per second on average (or equivalently, 1 task per 100 seconds per worker), this overhead is negligible. In the case of a long job with a huge number of tasks, such as a job with 50,000 tasks, each master will see a burst of task arrivals of size 250. This is in stark contrast with a distributed scheduler, who needs to generate and dispatch 50,000 tasks. This example clearly indicates that the resource demand on a master is modest and a single worker should be sufficient to serve as a master, which consumes only 1% (i.e., 1 out of 100) of the total worker resources in the cluster. This means that indeed, Pigeon is a highly scalable solution.

4.2 Performance evaluation

The number of workers in the whole cluster is used as a tunable parameter to adjust the load level. We use 50th, 90th and 99th percentile job slowdowns with respect to the case of *unlimited resources* (i.e., the case with zero communication time and zero task queuing time) for both short and long jobs as performance metrics. More specifically, the xth-percentile short/long job slowdown is defined as the xth-percentile short/long job completion time divided by the xth-percentile short/long job execution time. Here a job execution time is defined as the largest task execution time among all the tasks in the job.

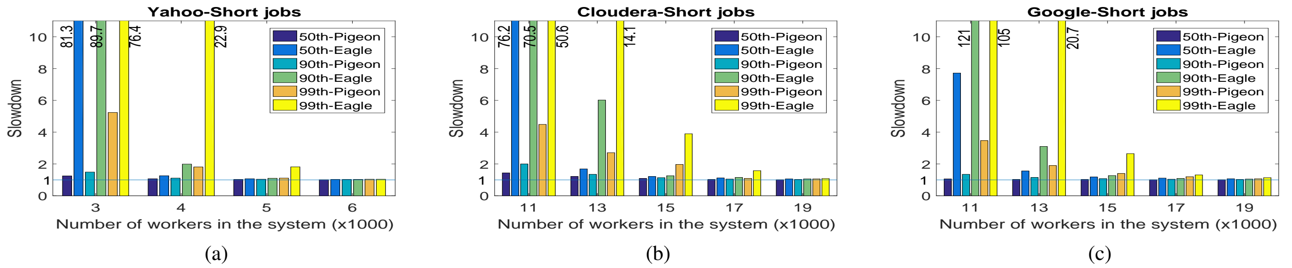
Figures 6 and 7 give the slowdowns of the 50th, 90th and 99th percentiles of short and long jobs for all the three traces. First, we note that at the fixed job arrival rate, as the number of workers in the cluster increases, the slowdowns of the two schedulers converge and approach 1 for both short and long jobs. This is expected, because as the cluster size becomes larger, or equivalently, the load becomes lighter, all the jobs experience smaller queuing delays and hence, smaller job completion times, regardless what scheduling mechanism is used. Hence, it is more interesting and important to focus on small cluster sizes or high load regions. As the cluster size reduces, we can see that remarkable performance gaps between the two emerge.

In the case of the Yahoo trace, at the cluster size of 3K, the slowdowns for short jobs in Pigeon are about 1.3, 1.5 and 5.3 times which indicates the queuing times are less than one job execution time for the 50th and 90th percentiles, and just above 5 job execution times for the 99th percentile. The results indicate that Pigeon achieves excellent short job performance even at very high cluster loads (about 95%). In contrast, the slowdowns for Eagle are above 70 times for all the three percentiles, implying that for Eagle, the queuing times are more than 70 job execution times for short jobs. Similar results can be found with the Google and Cloudera traces as shown in Figures 6(b)-(c). In what follows, we explain why Pigeon outperforms Eagle by such big margins.

Eagle improves over Hawk (detailed in [9] and not shown here) by allowing workers who are handling long jobs to reject the probes coming from distributed schedulers who handle short jobs. This allows a distributed job scheduler to issue more rounds of probes to discover workers that are not handling long jobs, hence alleviating the head-of-line blocking effect for short jobs. However, most lower priority (i.e., non-reserved) workers can still be blocked by the long jobs, either at high load or whenever a long job with a large fan-out degree arrives. In this case, after multiple rounds of random probing, most of the tasks from short jobs are forced to be served by the high priority (i.e., reserved) workers, which however, may become the bottlenecks themselves. For example, for the Yahoo trace, consider the case of a cluster with 3K workers and 60 high priority workers (2% as set in Eagle [9]) for short jobs. When a long job with 5900 tasks (i.e., the maximum number of tasks in a job for the Yahoo trace) arrives, each low priority worker has to serve, on average, about 2 tasks of the job. After the tasks of the long job are placed, all the upcoming short jobs following this long job are forced to be served by only 60 high priority workers after a number of rounds of probing. In other words, all the low priority workers are blocked by the long job, hence resulting in big job completion time for short jobs. The key difficulty is that as a hybrid scheduler, Eagle distributes tasks from short and long jobs independently by distributed and centralized schedulers, respectively.

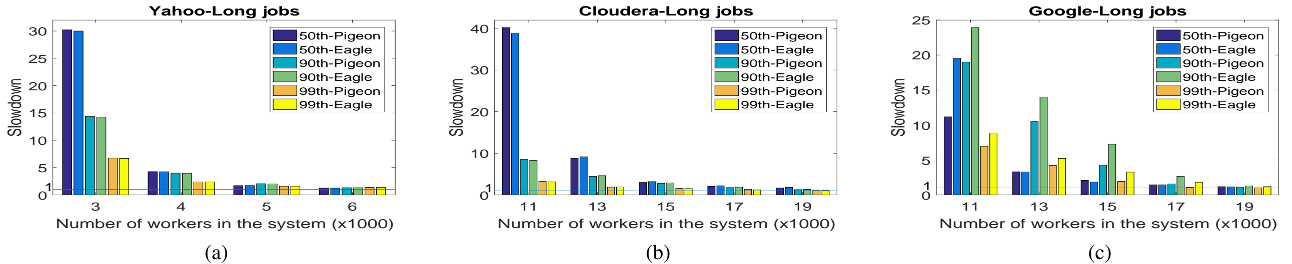
In contrast, Pigeon allows centralized scheduling of tasks coming from both short and long jobs and full resource sharing at the group level. This makes it possible for Pigeon to largely remove head-of-line blocking without starving the long job through weighted fair queuing and worker reservation. Again, consider the above example where a long job with 5900 tasks arrives at a cluster with 3K workers. Assume that the workers are divided into 30 groups of 100 each with 2 (i.e., 2%) workers reserved for the tasks from short jobs. Now about 197 (i.e., 5900/30) tasks from the long job are sent to each group. In a given group, the master dispatches as many tasks out of 197 to the available low priority workers as possible and the rest to the low priority queue, e.g., with 10 to the available low priority workers at the load of 90% (i.e., about 90% or 88 out of 98 are currently busy) and 187 queued. The upcoming tasks of short jobs are either served by an idle reserved worker or queued in the high priority queue. However, in addition to the 2 high priority workers, whenever a low priority worker becomes idle, it will first have high chance (19/20 at $W = 20$) to serve a task from the high priority queue. Since most of the long tasks (i.e., 187) are queued centrally at the master, not at the low priority workers, which however, is the case for Eagle, high priority tasks following these low priority tasks will not be blocked by the latter from accessing the low priority workers. Moreover, a task at the head of the high priority queue is likely to find an idle low priority worker soon, because the probability that one out of 98 busy lower priority workers will finish its task in the near future is high. This explains why Pigeon can significantly outperform Eagle in terms of short job performance, especially in the high load region.

The fact that Pigeon performs slightly better than Eagle even for long jobs, despite the use of the weighted fair queuing for short jobs over long ones, as depicted in Figure 7, can be explain as follows. First, Pigeon generally reserves a smaller number of workers for short jobs than Eagle (i.e., 9% vs. 17% and 8% vs. 9% in the cases



(a) (b) (c)

Figure 6: Short job completion time



(a) (b) (c)

Figure 7: Long job completion time.

of the Google and Cloudera traces, respectively and 2% vs. 2% in the case of the Yahoo trace), hence allowing more workers to be used by the long jobs. This explains why overall Pigeon outperforms Eagle in the cases of the Google and Cloudera traces but not as much in the Yahoo trace. Second, for all the real traces studied, the overall execution time for short jobs constitutes less than 20% of the total job execution time, implying that the possible negative impact of giving high priority to short jobs (i.e., letting W=20) on the performance of long jobs is quite limited.

The above results clearly demonstrate that Pigeon is a much more effective job scheduler than Eagle in terms of both design complexity (e.g., without probing phase, without having to run two different types of schedulers, and no worker involvement of scheduling) and performance.

5 Performance Evaluation on EC2 Cloud

In this study, we compare the performance of Pigeon against both Sparrow [19] and Eagle [9], the state-of-the-art distributed and hybrid job schedulers, respectively, in a small cluster on the Amazon EC2 cloud. The Pigeon implementation includes two parts: the Pigeon scheduler code and the Spark plug-in. Distributed Pigeon schedulers are concurrently deployed at the application frontends, exposing services to allow the framework to submit job scheduling requests using remote procedure calls (RPCs). All RPCs for internal communications between modules of a Pigeon scheduler are defined with Apache Thrift [1]. We directly run the available open source implementation codes for Sparrow [19] and Eagle [9]. m4.large instances are used to serve as workers, masters and schedulers.

The cluster is composed of 10 schedulers and 120 workers. For Pigeon and Eagle, 10% of the workers are reserved for short jobs. In Pigeon, the workers are divided into 3 groups with 40 workers each. One worker in each group is selected as a master and W is set to infinite (i.e., each master runs two strict priority task queues). A sample job trace including 5000 jobs is extracted from the Google trace. The task execution time is scaled to the range of 10ms to 100s and the job fanout degree is scaled to the range of 1 to 100. The short job cutoff time is set at 1s. It turns out that 10% jobs are long jobs, which however, consume about 88% overall task execution time, in line with the statistics of the original trace.

We use average job arrival rate as a tunning knob to adjust the cluster load. The job arrival process follows the Poisson distribution. The experimental results are also compared against the simulation results. The simulators for Pigeon and Eagle are the same as the ones described in the previous section and the open source event-driven simulator for Sparrow [19] is used.

We find that the short job performance for Sparrow and Eagle are very sensitive to the number of schedulers in use

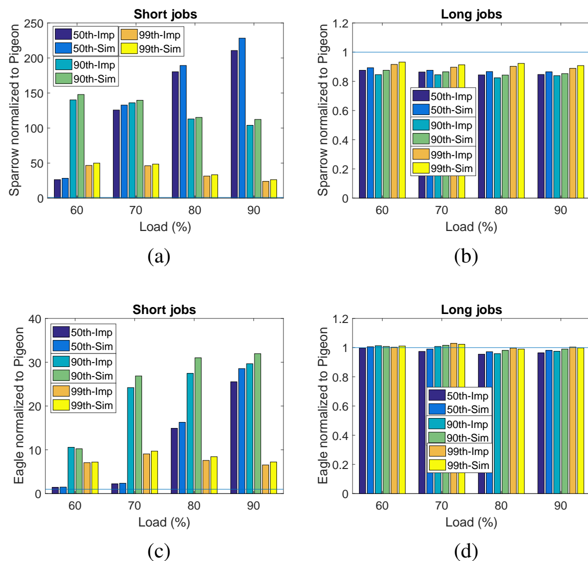


Figure 8: Experiment vs Simulation. Sparrow (a) short job and (b) long job; Eagle (c) short job and (d) long job.

(by changing the number of schedulers from 1 to 10). This is because the processing delay in the probe phase becomes non-negligible compared to the job execution time for short jobs. In contrast, Pigeon offers almost the same performance, regardless how many schedulers are used. In all the experiments, we use 10 schedulers to minimize the impact of the processing delay for Sparrow and Eagle.

Figure 8 depicts both measured (on EC2) (denoted as Imp) and simulated (denoted as Sim) 50th, 90th and 99th short and long job completion times normalized to Pigeon. The results for Sparrow and Eagle are depicted in Figures 8 (a) and (b) and Figures 8(c) and (d), respectively. Clearly, the experiment results are consistent with the simulation results. The differences between experiment and simulation are within 15% for short jobs and 5% for long jobs, mainly caused by the unaccounted processing overhead in the simulation.

As Sparrow does not distinguish between short and long jobs, it incurs up to 200 (10) times longer short job completion times than Pigeon (Eagle), although it offers up to 15% better long job completion times than both Pigeon and Eagle. This means that Sparrow is not effective in supporting short jobs in the presence of heterogeneous workloads. We also see that Pigeon provides significant performance gain for short jobs over Eagle. For example, at 90% load, the 50th, 90th and 99th percentile short job completion times for Eagle reaches about 25, 30 and 7 times longer than those for Pigeon. Pigeon and Eagle achieves comparable performance for long jobs at all cluster loads. The experiment results indicate that Pigeon is highly effective in handling heterogeneous jobs, which agrees with the simulation results obtained from the previous section.

Both the simulation and prototype implementation source codes will be made available in the public domain, upon the acceptance of the paper.

6 Practical Considerations

This section discusses some practical implementation issues, i.e., how to handle master failure and how to deal with heterogeneous workers and task assignment constraints.

6.1 Master Failure Recovery

A master plays a key role in a group. If a master fails, all the group information, such as the queued tasks and idle worker lists, are lost. In a full-fledged implementation of Pigeon, one may borrow a failure recovery mechanism widely used in the traditional distributed systems for failure recovery [24]. To allow fast recovery from a master failure, a second master is selected in a group. The second master can be another worker. A master needs to periodically update the second master on the group information and task state information. Whenever a master failure is detected, the second master can immediately take the master responsibility from the failed master. When a master failure happens, the second master sends a notice to each worker in the group and each scheduler in the cluster to notify them the changes, so that the subsequent tasks and idle worker notices are sent to the new master. Then a new second master should also be chosen for subsequent backups.

If both masters fail at the same time, the group information is lost. To quickly recover the group information, any worker in the group that detects such a failure can take the responsibility as a master. It broadcasts a message into the group to ask worker status. Each worker sends its response back to the new master with its status (idle or busy, priority, executing task, etc.). The new master also needs to send a message to each scheduler to get the task information sent to the group to recover the task queue list in the group. In case that multiple workers take the responsibility as a new master at the similar time, these workers can elect one as the new master based on some rules, e.g., the timestamp of master declaration time, CPU power or storage capacity and so on.

6.2 Dealing with Heterogeneous Workers and Tasks with Assignment Constraints

In the Pigeon design, we implicitly assumed that the same number of workers are assigned to each group and all the

workers have the same processing power. In practice, however, the number of workers in a group may not be conveniently set to be the same. Even if the numbers of workers assigned to different groups are the same, different workers may have different processing powers. Moreover, in practice, some tasks may have to be assigned to specific workers, as the needed resources or data are only available at those workers. All these may cause load imbalance among worker groups and hence have a negative impact on the performance of Pigeon. One possible solution is to require that all masters report their queue lengths for all the priority queues periodically to distributed schedulers. This will allow distributed schedulers to make more informed decision as to how to balance the load among groups.

7 Related Work

Today's dtacenter job schedulers can be classified into three categories, i.e., centralized, distributed and hybrid. The earlier job schedulers, e.g., Jockey[11], Quincy[15], Tetrished[26], Delay Scheduling[28] and Firmament[12] are centralized by design. A centralized scheduler can potentially provide high worker utilization, as it has a global view of the worker status for individual workers. But the scalability and head-of-line blocking are the major problems concerning centralized scheduling solutions. The scheduling decisions and status reports can overwhelm a centralized scheduler and cause additional job delay. Some shared-state schedulers, e.g., Apollo [2], Omega [22], and Mesos[13], use a centralized resource manager to maintain shared state. The job distributors are distributed but the decision maker is based on the shared status of the cluster resource availability. The shared status is updated by the distributed schedulers and/or workers. However, the shared state may not be always up-to-date and hence may result in job placement conflict and retries. This approach still requires a central entity for shared status maintenance. Recent work BigC [4] and Karios [8] propose to deal with job heterogeneity by suspending long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks.

Sparrow [19], on the other hand, is a fully distributed job scheduler based on random batch-based probe and late task binding. Although free from the scalability issues that plague the centralized job schedulers, the distributed schedulers and workers in Sparrow need to maintain fairly large amounts of task related state information and incur high communication cost for probing, including probe management, probe queuing, probe processing, and redundant probe removals. Furthermore, it does not perform well in highly loaded clusters nor in the presence of heterogeneous workloads. Another probe-based distributed scheduler, Peacock [18], organizes workers in a ring overlay network and a probe can be rotated to its neighbors at fixed time intervals to balance the probe queue lengths among workers. Peacock, however, requires that the workers communicate with each other to form and maintain a ring topology. Moreover, it inherits much of the drawbacks pertaining to probe-based solutions in general.

To solve the scalability issue while providing high performance in the presence of heterogeneous jobs, Hybrid schedulers [10, 9, 17, 27] are proposed. Hybrid schedulers combine a centralized scheduler and a set of distributed schedulers. Mercury [17] uses distributed schedulers to place jobs without latency requirement and a centralized scheduler to place jobs with guaranteed resource requirement. Hawk [10] uses a centralized scheduler for long job placement and the distributed schedulers for short job placement. The short job scheduling is similar to the techniques used in Sparrow, i.e., batch probing and late task binding based. Some workers are reserved to serve short jobs only, as a way to mitigate head-of-line bocking. Moreover, an idle worker can steal tasks belonging to short jobs from other workers to improve efficiency. Eagle [9] improves over Hawk by introducing sticky batch probe with each probe staying on a worker until all the tasks of the job finish. It also allows multiple rounds of probing to mitigate head-of-line blocking. These hybrid schedulers need a central scheduler that can still pose a potential bottleneck. Moreover, short job scheduling is still probe-based and hence, inheriting its shortcomings.

More complex queuing mechanisms than priority queuing are being used to improve the job performance. Queue reordering [9, 14, 26, 22] is used to reduce the job completion time. More complex queue management techniques [20] such as appropriate queue sizing, prioritization of task execution via queue reordering, and starvation freedom are also being used to improve the efficiency of job scheduling.

8 Conclusions

In this paper, we propose Pigeon, a distributed job scheduler for datacenters. In Pigeon, workers are grouped. Each group has a master worker which centrally manages all the tasks handled by the group. Weighted fair queuing is used to provide priority service differentiation between tasks of short jobs and tasks of long jobs. A small portion of workers in each group are reserved to serve high priority tasks only. The ability of each master in managing its group resources centrally makes Pigeon highly effective in scheduling heterogeneous jobs. The analysis, simulation and experiment results demonstrate that Pigeon out-

performs Sparrow and Eagle by significant margins.

9 Acknowledgments

This work is supported by the US NSF under Grant No. XPS-1629625 and SHF-1704504.

References

[1] Apache thrift. *https://thrift.apache.org/*.

[2] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of OSDI* (2014).

[3] BRUTLAG, J. Speed matters for google web search. In *Google* (2009).

[4] CHEN, W., RAO, J., AND ZHOU, X. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proceedings of USENIX Annual Technical Conference* (2017).

[5] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of VLDB Endowment* (2012).

[6] CHEN, Y., GANAPATHI, A., GRIFFITH, R., AND KATZ, R. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of MASCOTS* (2011).

[7] COOPER, R. B. Introduction to queueing theory. *North Holland* (1981).

[8] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates . In *Proceedings of ACM Symposium on Clod Computing (SOCC)* (2013).

[9] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Job-aware scheduling in eagle: divide and stick to your probes. In *Proceedings of ACM Symposium on Clod Computing (SOCC)* (2016).

[10] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2015).

[11] FERGUSIN, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of EuroSys* (2012).

[12] GOG, I., SCHWARZKOPF, M., A., G., EATSON, R. N. M., AND HAND, S. Firmanent: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of USENIX Symposium on Iperating System Design (OSDI)* (2016).

[13] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI* (2011).

[14] HUNG, C.-C., GOLUBCHIK, L., AND YU, M. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of SoCC* (2011).

[15] ISARD, M., PRABHAKARAN, V., CURREY, J., UDI, W., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of SOSP* (2012).

[16] JEON, M., KIM, S., HWANG, S.-W., HE, Y., ELNIKETY, S., COX, A. L., AND RIXNER, S. Predictive Parallelization: Taming Tail Latencies in Web Search. In *Proceedings of ACM SIGIR* (2014).

[17] KARANASOS, K., RAO, S., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2015).

[18] KHELGHATDOUST, M., AND GRAMOLIM, V. Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queuess. In *Proceedings of International Conference on Parallel and Distributed Computing* (2018).

[19] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of ACM Symposium on Operating System (SODP)* (2013).

[20] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient Queue Management for Cluster Scheduling. In *Proceedings EroSys* (2016).

[21] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC* (2012).

[22] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys* (2013).

[23] SHELDON, R. Introduction to probability models. *Academic Press* (2014).

[24] STUTSMAN, R. S. Durabilit and Crash Recovery in Distributed In-Memory Storage Systems . In *Dissertation of Doctor Philosophy* (1987).

[25] SUO, K., RAO, J., JIANG, H., AND SRISA-AN, W. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of ACM European Conference on Computer systems (EuroSys)* (2018).

[26] TUMANOV, A., ZHU, T., PARK, J. W., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of EuroSys* (2016).

[27] XIA, Y., REN, R., CAI, H., VASILAKOS, A. V., AND LV, Z. Daphne: A Flexible and Hybrid Scheduling Framework in Multi-Tenant Clusters. *IEEE Transactions on Network and Service Management 15*, 1.

[28] ZAHARIA, M., BORTHAKUR, D., SARMA, J., KHALED, E., SHENKER, S., AND STOICA, I. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of EuroSys* (2010).