Phoenix: A Constraint-aware Scheduler for Heterogeneous Datacenters

Prashanth Thinakaran*, Jashwant Raj Gunasekaran*, Bikash Sharma†, Mahmut Taylan Kandemir*, Chita R. Das*
* Computer Science and Engineering, Pennsylvania State University
† Microsoft Corporation
{prashanth,jashwant,kandemir,das}@cse.psu.edu, bsharma@microsoft.com

***Abstract*—Today's datacenters are increasingly becoming diverse with respect to both hardware and software architectures in order to support a myriad of applications. These applications are also heterogeneous in terms of job response times and resource requirements (eg., Number of Cores, GPUs, Network Speed) and they are expressed as task constraints. Constraints are used for ensuring task performance guarantees/Quality of Service(QoS) by enabling the application to express its specific resource requirements. While several schedulers have recently been proposed that aim to improve overall application and system performance, few of these schedulers consider resource constraints across tasks while making the scheduling decisions. Furthermore, latency-critical workloads and short-lived jobs that typically constitute about 90% of the total jobs in a datacenter have strict QoS requirements, which can be ensured by minimizing the tail latency through effective scheduling.**

In this paper, we propose Phoenix, a constraint-aware hybrid scheduler to address both these problems (constraint awareness and ensuring low tail latency) by minimizing the job response times at constrained workers. We use a novel Constraint Resource Vector (CRV) based scheduling, which in turn facilitates reordering of the jobs in a queue to minimize tail latency. We have used the publicly available Google traces to analyze their constraint characteristics and have embedded these constraints in Cloudera and Yahoo cluster traces for studying the impact of traces on system performance.

Experiments with Google, Cloudera and Yahoo cluster traces across 15,000 worker node cluster shows that Phoenix improves the 99^{th} percentile job response times on an average by 1.9× across all three traces when compared against a state-of-the-art hybrid scheduler. Further, in comparison to other distributed scheduler like Hawk, it improves the 90^{th} and 99^{th} percentile job response times by 4.5× and 5× respectively.

***Keywords*—Scheduling; Hybrid; Heterogeneous Data Center; Constraint-aware; Resource Management; Performance**

I. Introduction

With the emergence of cloud computing, datacenters are getting increasingly diverse both in terms of the hardware and system software stack. Datacenter schedulers play an important infrastructure component role in such cloud environments for efficient matching between application demands and available cluster resources. A well-architected scheduler with optimized resource management policies has direct bearings on reducing operational and capital expenditures (CapEx and OpEx) [1] of datacenters as it boosts the utilization of resources leading to reduction in the number of machines and datacenter resource requirements to support the applications demands. Today's

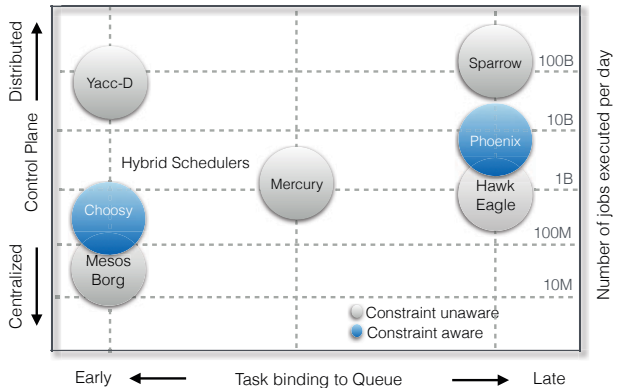


Fig. 1: Design space of datacenter schedulers.

Cluster schedulers are expected to take advantage of the growing heterogeneity in datacenter infrastructure to perform smart workload placements. The applications are also having diverse needs in terms of their Quality of Service, placement preferences, and also require special hardware like GPUs, FPGAs, Storage type (SDD,HDD), etc,. [2]–[7]. This trend is expected to grow as we see more applications take advantage of application specific hardware accelerations like web search [8], Memcached [9, 10] and convolutional neural networks [11] using FPGAs. Thus, constraints provide a mechanism to take advantage of hardware heterogeneity by the applications. They also enhance scheduler effectiveness in better matching applications requirements with cluster hardware resources. This results in improved resource utilizations and faster job response times. Constraints also aid in specifying job placement preferences to ensure fault tolerance (eg.,Multiple jobs of the same application are spread out across racks) and application level optimizations (micro-architecture, compiler version, etc).

To meet the diverse needs of such applications, the architecture and design of cluster schedulers have evolved considerably. Specifically, there have been multiple variants of scheduler designs – monolithic, disaggregated, distributed, etc., each with its own advantages and disadvantages [16, 21]. Existing schedulers can be broadly classified based on the scheduler logic design and the time when a task commits itself to a worker queue as shown in Figure 1. The volume of the scheduling decisions heavily depend on such design choices. Schedulers like Mesos [13] and Borg [12] are designed to be

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.262

Scheduler	Control Plane	Binding	Queuing	Queue Reordering	Load Balancing	Placement constraints
Borg [12]	Hierarchical	Early	Global	✗	Static	✗
Mesos [13]	Hierarchical	Early	Global	✗	Static	✗
Paragon [14]	Monolithic	Early	Global	✗	Static	✗
Sparrow [15]	Distributed	Late	Worker side	✗	Static	Trivial
Hawk [16]	Hybrid	Late	Worker side	✗	Stealing	Trivial
Eagle [17]	Hybrid	Late	Worker side	SRPT	Stealing	Trivial
YacC+D [18]	Hybrid	Early	Both	SRPT	Adaptive	✗
Tetrisched [19]	Monolithic	Early	Global	✗	Static	Trivial
Choosy [20]	Hierarchical	Early	Global	✗	Static	Single resource
Phoenix	Hybrid	Late	Worker side	CRV based	Adaptive	Multi resource

TABLE I: Comparison of Phoenix and other contemporary datacenter schedulers.

hierarchical (see Table I). These schedulers suffer from the following limitations:

1) Their control plane is centralized and does not scale along with the resources under high load/contention scenarios.
2) Production schedulers are frequently subjected to routine maintenance and updates, and in case of a failure of one scheduler node, it could cascade and affect the whole datacenter operations.
3) They also bind their tasks early to the worker queue, thus losing the flexibility of task migration and are ill suited for the class of short-lived interactive applications which dominate (80-90% [22]) the datacenter.

On the other side of the spectrum, schedulers like Sparrow [15] are completely distributed and the control plane is disaggregated. They are limited by:

1) Lack of capabilities to handle today's complex application scheduling requirements.
2) Insufficient information on the status/load of worker nodes.
3) Agnostic of any interference from co-located applications.
4) Poor job response around times for short tasks due to head of the line blocking.

In recent times, hybrid scheduler design has gained prominence as it improves upon the pitfalls of other contemporary designs. The recent class of schedulers like Hawk [16] and Eagle [17] combine the respective advantages of both centralized and distributed schedulers. Table I provides a comprehensive summary of design choices of existing schedulers. Most of the hybrid and distributed schedulers make use of techniques such as Shortest Remaining Processing Time (SRPT), Sticky Batch Probing (SBP) and task sampling, which do not adapt well to multiple placement constraints scenario as they are agnostic of job placement constraints. SRPT based task reordering does not always result in faster job response times as the delay experienced by tasks asking for the constrained resources vary based on the resource availability. SBP and task sampling are poor estimators for queue waiting times while scheduling for jobs with resource constraints. We further discuss this detail in Section III-C.

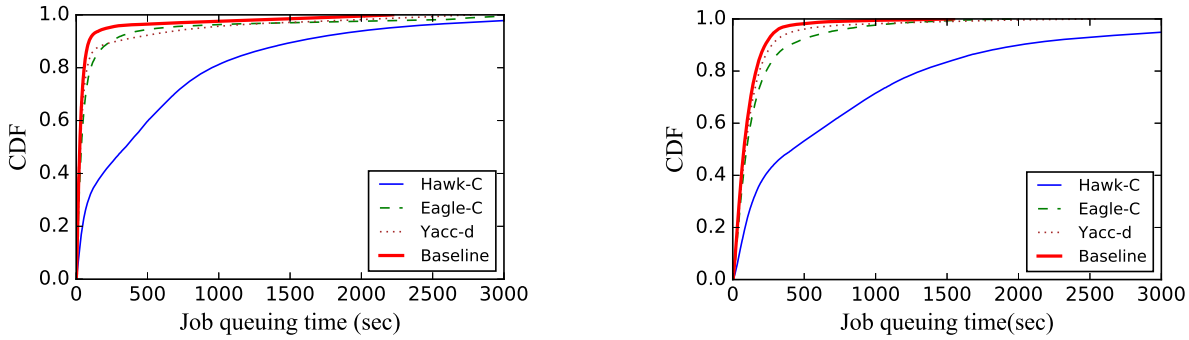
Though prior works like Tetrisched [23] and Choosy [20] have addressed the task scheduling issues in presence of constraints, they are built around Yarn [24] or Mesos [25], which operates with a global job placement queue with slots-based centralized logic as seen in Figure 1. However, their design choices does not scale to the demands of the growing class of latency-critical applications like interactive web services and streaming analytics-based queries.

Thus, in this paper, we propose a constraint-aware scalable scheduler, called Phoenix[1]. The salient features of Phoenix is described in the last row of Table I, it is a hybrid scheduler with late binding of tasks to worker queues. It uses a novel Constraint Resource Vector (CRV) based task reordering mechanism across constrained queues to holistically improve the job turnaround times. Constraint Resource Vector (CRV) is defined as a vector of node resources like <cpu, mem, disk, os, clock, net_bandwidth>. We use CRV demand (tasks asking for the resource) and supply (total available resources) ratio of constrained resources to determine the estimated queuing delay and an M/G/1 queuing model is used to dynamically reorder tasks based on its CRV ratio values to improve the overall job turnaround times.

We make the following contributions in this paper:

1) We propose a constraint-aware hybrid scheduler that does dynamic task reordering using a novel Constrained Resource Vector (CRV) based node monitor. Phoenix is developed on top of the open source Eagle scheduler and is extended to handle constraint specification of jobs.
2) We employ a proactive admission control by negotiating resources for tasks in which all the constraints could not be satisfied. This is achieved by succinct sharing of demand-supply information of the available resources.
3) We develop a queuing model to estimate the waiting times of highly contentious resources. We use this information to reorder tasks in the worker queue leading to improved job turn around times.
4) We analyze the open source Google traces to characterize different types of constraints and embed these constraints to other public traces from Yahoo and Cloudera for enabling constraint-aware scheduling study.
5) We conduct experiments with Google, Cloudera and Yahoo traces and demonstrate that Phoenix can improve the 99^{th} percentile job completion times on an average by 1.9x and 5x over constraint aware Eagle (Eagle-C) and constraint aware Hawk (Hawk-C), respectively by reducing the tail

[1]Phoenix is a mythological bird that is cyclically regenerated. Since the scheduler improves over its predecessors, attaining a new life.



(a) Yahoo trace with constraints.

(b) Cloudera trace with constraints.

Fig. 2: Job queuing times for two different production traces with task placement constraints

latency of short jobs. Phoenix does not affect the fairness and the execution times of the other long and unconstrained jobs across the cluster.

II. Background and Motivation

A. Constraint Based Job Requests in Cloud Schedulers

Datacenters are heterogeneous in terms of CPU (eg., ISA, clock frequency, number of cores), presence of accelerators (eg., coprocessors, GPUs, FPGAs), memory (eg., bandwidth, capacity), network (eg., bandwidth, technology) and storage (eg., capacity, technology, redundancy) configurations. For example, a job may request for two server nodes belonging to x86 ISA with a network speed of 1 Gbps between them. Cloud vendors allow tasks to subscribe to a combination of heterogeneous resources using task constraints. Recent studies show that the tasks that subscribe to different constraints in a production datacenter account for more than 50% of all the tasks [2, 26]. Thus, any scheduler that is aware of the constraints in jobs can substantially benefit from using the information in its scheduling decisions.

B. Impact of Constraints in Existing Systems

To understand how constraints are used in reality and study their relative importance, we summarize relevant information from Google trace [27] and Utilization Multiplier [2] in Table II. the second column shows the relative slowdown in various jobs requesting for a specific constraint w.r.t to a no-constraint job. As can be seen, the trace runs for $\approx 25M$ jobs and 80% these jobs suffer a $\approx 2\times$ slowdown waiting for a server node of the requested ISA/CPU cores/network speed to become available. Also, a slowdown of $1.7\times$ can be seen for other resources or machine properties like OS kernel, CPU frequency etc., although such requests do not dominate this trace. The results indicate that constraint awareness in scheduling can potentially mitigate the job slowdowns that might have occurred due to poor placement policies.

C. Constraint-Awareness in Existing Systems

We now discuss two classes of schedulers, namely, centralized and distributed schedulers (as shown in Table I) to understand how constraints are handled in the traditional systems.

1) Centralized Schedulers: Centralized schedulers such as Hadoop fair scheduler [28], the Capacity scheduler [29], Yarn [24], Choosy [20] and Tetrisched [23] uses slot-based models as a simplification to denote all resources as a homogeneous set. But as noted in Dominant Resource Fairness (DRF) [30], (i) these centralized schedulers are inefficient for allocating/managing multiple fine-grained resources; and (ii) their centralized control plane becomes an overhead in scaling along with greater volume of job requests/constraints.

2) Distributed and Hybrid Schedulers: The state of the art hybrid schedulers (e.g., Eagle [17]) does SRPT based task scheduling to improve the overall job turn around times at the worker queues. In case of jobs coming in with multiple types of resource requests in form of constraints each machine might have different utilization rates. In such cases, SRPT may not be as effective as it is for a single homogeneous resource (e.g. CPU). This is illustrated in Figure 3, where the inter-arrival patterns of the jobs are highly sporadic with valleys and peaks. Generally during high loads, the peaks significantly contribute to the tail latency of a short job's completion times. It is observed that, the job queuing delay of constrained tasks cascade its delay into that of subsequent job's completion times. When this happens, it takes a long time to reach the same QoS state as seen in Figure 3 the "unconstrained" execution.

This trend is common across all the existing schedulers like Hawk [16] and Sparrow [15] for different production traces well. It is seen from Figures 2b and 2a, the job queuing times for two different production traces Cloudera and Yahoo respectively. The baseline is the task queuing delay in case of jobs without constraints. Hawk-c scheduler incurs heavy queuing delays across all the percentiles of task run times. The tasks scheduled by Eagle-C and Yacc-d experience 2 to $2.5\times$ task queuing delays because of constraints.

In summary, just the SRPT based queue reordering would not improve the overall turn around times of job's with tasks which has specific resource constraints as the demands for various resources vary across different jobs. Also, the high volume of requests make a strong case for distributed-scheduler instead

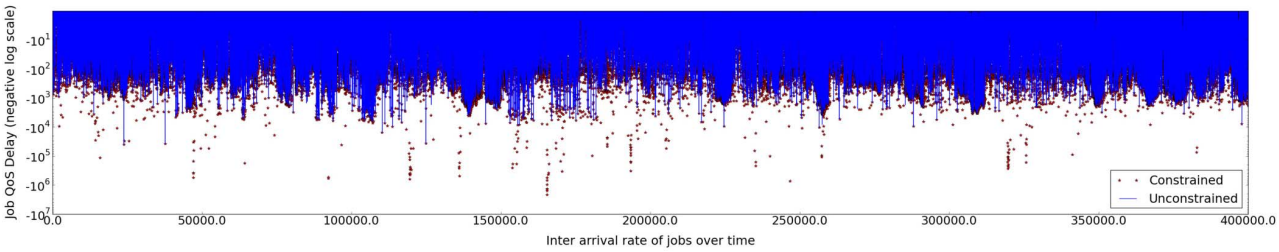


Fig. 3: Google trace executed in Eagle-C. Queuing delays of constrained and unconstrained jobs over time.

Task Constraints	Relative Slowdown	% Share	Occurrence
Architecture (ISA)	2.03×	80.64	20412140
Number of Nodes	1.96×	0.28	71103
Ethernet Speed	1.91×	0.18	30128
Number of Cores	1.90×	18.28	2856749
Maximum Disks	1.90×	8.57	1665117
Kernel Version	1.77×	0.21	52722
Platform Family	1.77×	0.05	14473
CPU Clock Speed	1.76×	0.16	42688
Minimum Disks	0.91×	0.66	168656

TABLE II: Distribution of constraints in as reported in Google cluster traces [27]; Relative Slowdown: Slowdown of a constrained job w.r.t an equivalent but unconstrained job.

of a fully centralized logic. Since latency critical tasks account for ≈90% of the total requests, the scheduler proactively needs to be aware of the expected delay due to the individual task constraint or combination of constraints (and hence infer the utilization of various resources) to schedule tasks.

Hence, there is a need for a scalable scheduler that could handle tasks with multiple task constraints without compromising on the QoS by dynamically adapting itself to the changing demand to supply ratio of constrained resources.

III. Modeling and Synthesizing Constraints

A. Classification of Constraints

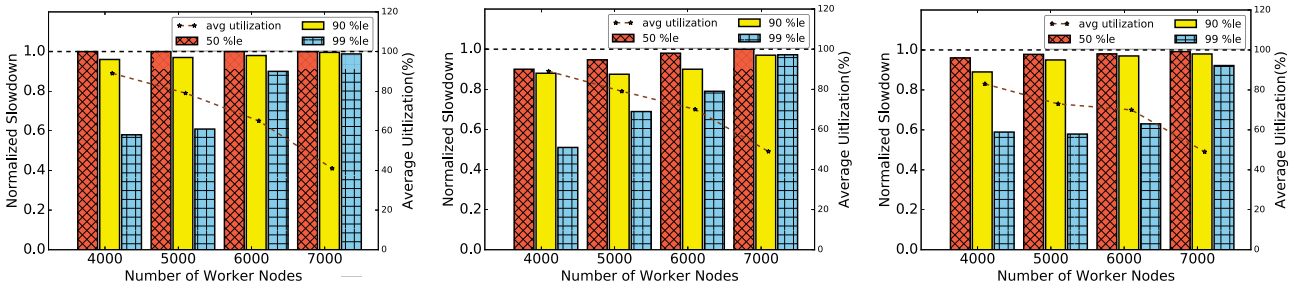
Constraints can be broadly classified into three categories: hard, soft, and placement constraints. A job can potentialfly have one or more constraints of any category and typically the hard constraints are strict requirements without which a job cannot run (eg., number of CPUs, minimum memory, requirement for a machine with public IP address). On the other hand, soft constraints (eg., CPU clock speed, network bandwidth) can be relaxed or negotiated by trading off the job's performance. The various constraints from Google datacenter traces [27] are enlisted in Table II. The placement constraints or combinatorial constraints are affinity preferences of group of tasks of a particular application like Hadoop or Spark that prefer to be scheduled close to each other due to data locality reasons. Few applications might prefer its tasks to spread out across on multiple racks for fault tolerance guarentees (eg., disk failure or network switch failure at a node or a rack might bring down the whole progress of the application, hence independent tasks are spread out). These tasks include constraints in terms of rack_id. For example, Mesos [13] allows jobs to specify its locality preferences. But they do not have provisions for tasks to avail for different heterogeneous resources in the datacenter. These affinity constraints have a significant impact on task scheduling delay by a factor of 2 to 4 times [22].

B. Synthesizing Task Constraints

We make use of the publicly available Google's cluster workload traces [27], which is a month-long production trace collected across 12,500 nodes. We do not go over the details of the job heterogeneity and task distributions as it has been analyzed by prior works [2, 22] in detail. Google has obfuscated its original values before making it public. The task demands in form of constraints and the associated values are also hashed in the traces. Hence, we use the constraint frequency vector of Google cluster-C from the paper [2] which gives the frequency distribution and semantics of each individual constraint and correlated it with the constraint frequency distribution of the Google cluster trace [27]. We use this to augment the hashed entries with real constraint attributes and values, which is given in Table II.

In order to synthesize constraints for the other production traces like Yahoo and Cloudera, we use the benchmarking technique proposed by Sharma et al., [2] to characterize and incorporate constraints in to Yahoo and Cloudera traces. We further cross validate our model with the task and machine constraint occurrence fraction from [2] to ensure the correctness of the correlation. Thus, we synthesize representative constraints for workload traces like Yahoo and Cloudera other than Google for evaluation. This model is representative of the distribution of different task constraints in a production datacenter. It is claimed in the paper [2] that the model accuracy is close to 87% in case of Google-C [27] cluster since the model was trained on statistics collected from the traces across the three biggest clusters in Google (Cluster-A, B, C).

We modify Sparrow, Eagle and Hawk schedulers to handle jobs with constraints in production traces like Yahoo, Cloudera and Google. We refer to this version as Sparrow-C, Eagle-C and Hawk-C respectively, whose design specifications are detailed in the results section. It is seen from Figure 4 that 99^{th} percentile of job response times has gone up uniformly across all the traces by an average of 1.7× in case of Eagle-C. This trend worsens when the utilization of the data center goes up. As the inter arrival rates of short jobs surge, schedulers like Sparrow-C [15] and Hawk-C [16] perform even worse than Eagle-C, in such cases where we observe an average of 20× slowdown in job response times in 99^{th} percentile



(a) Yahoo (b) Cloudera (c) Google

Fig. 4: Short job response times for unconstrained jobs normalized to constrained jobs for 50^{th},90^{th},99^{th} percentiles scheduled with Eagle-C.

job response times across all the workloads. State-of-the-art distributed and hybrid schedulers do not adapt to these sudden surges of constrained jobs since the schedulers are agnostic of the expected delay due to individual task constraints.

C. Requirements for Scheduling Constrained Jobs

1) Job response times: The strategy that most of the hybrid schedulers [17, 18] use in improving the over all job response times is SRPT. Reordering tasks based on shortest remaining processing times of individual tasks reduces straggler tasks thus mitigating the tail latency. But naive SRPT has fairness issues and is an overkill at high loads as it leads to starvation of other longer jobs or jobs with diverse task run times. Greedily prioritizing the task with least delay would benefit intermittently but would not contribute to the overall average job response times and comes at the cost of fairness of the other unconstrained jobs. The scheduler needs to intelligently adapt its logic based on the dynamic supply demand surges of constrained resources.

2) Admission Control: Recall that a job could arrive with one or more constraints. The scheduler has to negotiate resources and satisfy these constraints even at peak congestions across all the worker queues. There could be situations where two out of three constraints of the job could be satisfied in which, a scheduler could relax in case of soft constraints with performance trade offs. Schedulers like Choosy [20] fail to capture such scenarios of multi-resource constraints.

3) Late Binding: Job response time is a cumulative factor of queuing time, placement latency and algorithm execution time. To avoid the task being committed to a single machine, schedulers like [15]–[17] use delayed probing which places probes instead of tasks. Batch sampling [15] and probe aggregation of jobs [17] are used to predict the queues with least lengths. Note that, these techniques are not always the accurate estimators of queue waiting times (see Equation 1), as the queues with least lengths do not always correlate with least turnaround times. Thus, distributed schedulers need a low overhead queue waiting time estimator in order to make near optimal scheduling decisions when compared to fully centralized scheduler.

4) Load Balancing: Production schedulers are expected to improve the overall resource utilization of the datacenter. They employ load balancing techniques such as work stealing from idle workers. This becomes challenging when the scheduler is distributed because it is agnostic of the load and surges across the entire cluster. Stealing based techniques fail to provide accurate scheduling decisions due to the fact that not all the tasks could be relocated or stolen as they might have resource specific constraints. Also task migration has its own overheads as it might not to fit within the task's SLA guarantees. As a result, a constraint aware hybrid scheduler is needed to provide optimal task placements at sub-millisecond latency. We propose Phoenix, which overcomes the above challenges by proactive admission control and CRV based queue reordering described in the following section.

IV. Phoenix Architectural Overview

A. Phoenix Constraint-aware Scheduler

Phoenix is a heterogeneity and constraint-aware hybrid scheduler, which does QoS aware task placements and re-ordering to the queues at the worker nodes. The scheduler is aware of both job heterogeneity (long or short jobs) and node heterogeneity (architectural). It uses centralized scheduler for scheduling long jobs and distributed schedulers for short jobs, which interacts with `CRV_Monitor` shown in Figure 5 and keeps track of the Constraint Resource Vector (CRV) ratio of individual constraints, which is used in dynamic reordering of constrained tasks. Phoenix improves over the existing hybrid schedulers by introducing the `CRV_Monitor` unit as shown in Figure 5 which monitors the resource supply and demands from incoming constrained jobs every heartbeat cycle. Each individual worker queue can satisfy one or more constraints hence these queues are inherently heterogeneous. The CRV of a node is a vector of node resources represented as `<cpu, mem, disk, os, clock, net_bandwidth>`. For every node, the `CRV_Monitor` calculates the ratio of demand and supply for every constraint per heartbeat interval and updates the `CRV_Lookup_Table`. Using this information we further estimate the waiting time ($E[W]$) for every queue using Pollaczek-Khinchin *M/G/1* queuing theory [31] model

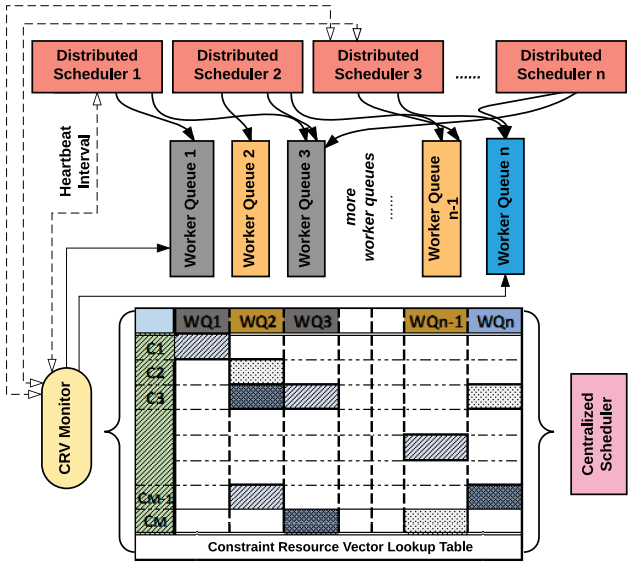


Fig. 5: Overview of job scheduling in Phoenix.

based estimator (Equation 1). The first part of the dot-product directly correlates to the load at the constrained worker queues and the second part represents the variability in estimated task execution time. To minimize the waiting time at every worker queue, both the load and variance in estimated task execution times should be minimal. Since we assume that each slot has independent task queues for execution *M/G/1* model is sufficient to estimate the queue waiting times. Phoenix uses centralized scheduler for long jobs and distributed scheduler for short jobs and thus, reducing the variance of the estimated task execution times at any worker node. This minimal variance guarantees the stationary assumptions of P-K Equation 1 valid, ensuring the accuracy of the estimator. In Equation 1, ρ is the worker load collected periodically from all workers for every heartbeat interval and $E[S]$ is expected service time for the tasks in that particular worker. $E[S]$ of a worker is a combination of the execution time of the currently running task and the cumulative sum of estimated individual task execution time [31]. In other words $E[S]$ is $(1 \div \lambda)$ and ρ is $(\lambda \div \mu)$ where λ is the average inter arrival time and μ is the average service time.

$$E[W]^{M/G/1} = \frac{\rho}{1-\rho} \cdot \frac{E[S^2]}{2E[S]} \quad (1)$$

Phoenix estimates waiting times ($E[W]$) values for each worker queues for every heartbeat interval. Also, in the same interval, we proactively warmup the `CRV_Lookup_Table` with CRV values. During peak resource demands, when the CRV values increase beyond a fixed threshold (`CRV_Threshold`), Phoenix initiates the `CRV_Based_Reordering` algorithm 1. Otherwise, Phoenix does SRPT based reordering. This is because for any c.f.m.f.v (continuous, finite mean, finite variance) distributions with heavy tail property, at least 99% of the jobs have a lower response times under SRPT than any other scheduling techniques (eg., FIFO, SJF) when ρ<0.9 [31]–[33]. Therefore, Phoenix opportunistically adapts itself to the CRV based reordering from SRPT during peak loads. Reordering based on the CRV values on congested worker queues curtails the tail latency as it significantly reduces the execution time of straggler tasks. Phoenix does not consider the utilization ρ as the overall datacenter rather for each type of resource as mentioned in the Table II achieving fine grain heterogeneous resource management. In case of tasks over subscribing a particular resource type it is quite for ρ to be over 0.9 during bursts.

Phoenix uses late binding for flexible task placements in order to avoid being committed to a queue early on. When a constrained task arrives, the corresponding proxy task probes are sent to the worker nodes that could satisfy the constraints. The probable nodes to which the probes are sent is decided by light weight bit vector message exchanges across the distributed schedulers similar to Eagle's Succint State Sharing (SSS) technique [17]. The `CRV_Monitor` aggregates the potential list of worker nodes that could satisfy the constraints that tasks demanded. For example, a job with 10 tasks with an *ISA* and *Kernel_version* constraints, if the probe ratio is set to two, then 20 probes are sent to the probable list of nodes that satisfies both the constraints.

Algorithm 1 CRV-based reordering

```
1:  procedure CRV_MONITOR(Queue)
2:      while every Heartbeat interval do
3:          for worker in all_workers do
4:           worker(E[W]) ← Estimate_Waiting_Time(CRV)
5:          end for
6:          for all Constraints
7:             CRV_Lookup_Table(Constraint)
8:          for worker in all_workers do
9:              if worker(E[W]) > Qwait_threshold then
10:                 CRV_based_reordering(Queue)
11:             end if
12:         end for
13: procedure ESTIMATE_WAITING_TIME
14:     for task inqueued_tasks
15:         μ ← Avg(last_serviced_tasks)
16:         λ ← Avg(inter_arrival_rate)
17:         E[W] ← Equation1
18:     end for
19: procedure CRV_BASED_REORDERING(QUEUE)
20:     Max_CRV ← getMax(CRV)
21:     for task in Queue.tasks[CRV] > Max_CRV
22:         task.slack++
23:         if task.slack < Slack_theshold then
24:             Reorder_Task(task)
25:         end if
26:     end for
```

B. Algorithm Implementation

As explained above, the utilization of the resources in terms of constraints is periodically maintained at the `CRV_Monitor`.

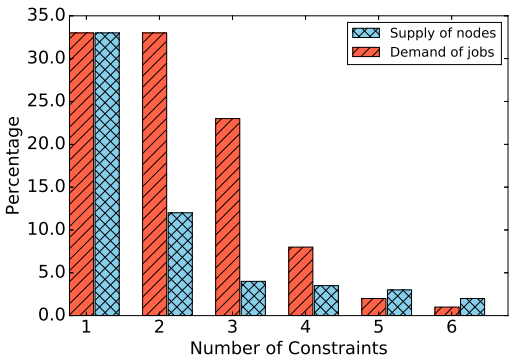


Fig. 6: Constraints supply/demand distribution.

As a result, Phoenix constantly monitors the number of constrained and unconstrained tasks along with the impact of delay on tasks due to these constraints. This periodic synchronization of information across the distributed schedulers poses an overhead. Phoenix mitigates this by following: First, having the CRV vector available offline. Second, the distributed scheduling decisions are loosely coupled with `CRV_monitor` every periodic heartbeat intervals ensuring the scheduler being synchronized with the `CRV_monitor`. Third, Phoenix improves the job completion times by reordering tasks based on their CRV values.

The algorithm 1 uses the `CRV_MONITOR` procedure to govern the CRV ratio of every constraint. When this ratio goes beyond a set `CRV_threshold`, we mark all the workers whose $E[W]$ is greater than `Qwait_threshold`. For those workers which are marked, Phoenix invokes the `CRV_based_reordering` function to reorder the tasks based on CRV values. We conservatively set the `Qwait_threshold` which translates to peak utilization in the datacenter. The `CRV_threshold` for each constraint is determined based on the task execution statistics given in the Table II. `Max_CRV` function calculates the maximum of the values in CRV vector and reorders the tasks having the same `MAX_CRV` value. We also implemented a starvation threshold (`Slack_threshold`) which ensures the fairness among all tasks. The `Slack_threshold` limits the number of times a task can be bypassed (preempted) because of other reordered tasks) within a queue. The complexity of the algorithm is split into two parts. First, for every heartbeat interval, the cost of waiting time estimation is `O(n)`, where `n` is the number of workers. Since they could be calculated in parallel for all workers, thus reducing the complexity. Second, during peak utilization where the $E[W]$ shoots beyond `Qwait_threshold`, `CRV_based_reordering` is invoked. The complexity of `CRV_based_reordering` is `O(p)`, where `p` is the number of probes per worker queue. This is the same complexity of executing SRPT based reordering during median utilizations.

V. Evaluation Methodology

A. Implementation

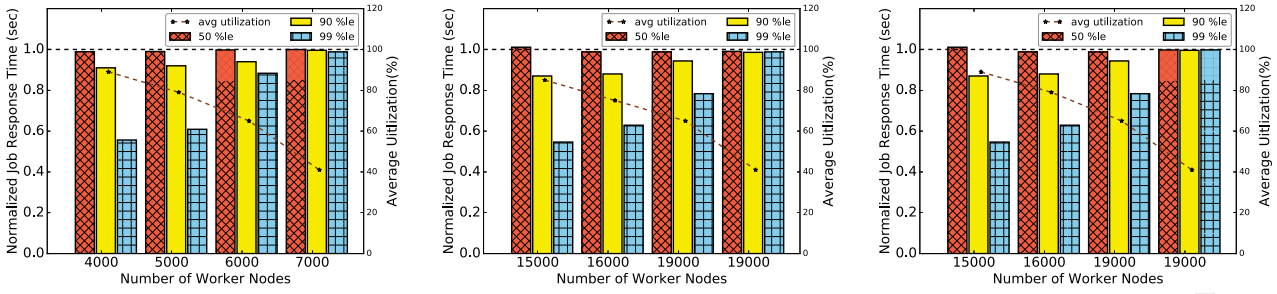
We implemented Phoenix on a trace-driven simulator which is used to evaluate Eagle [34] and Sparrow [35]. At each worker node, there is one slot for execution and a queue for tasks waiting to be executed. The publicly available cluster trace from Google [27] is used as the primary trace for evaluation. In addition, we also use Yahoo and Cloudera traces from [36, 37]. We observed that the cluster load is bursty and unpredictable with the peak to median ratio ranging from 9:1 to 260:1 in these traces. In general the task execution times are Pareto bound, where short jobs constitute of 80% to 90% of the total jobs. From the Table III, which gives constraint distribution of the jobs across all three traces. It is observed that approximately 50% of the tasks has one or more constraints.

Google cluster trace has task constraints with attribute names and values. In addition to it, We synthesize task placement constraints for Yahoo and Cloudera traces based on the model described in Section III-B. On average, 50% of the tasks are constrained and every job is heterogeneous with multiple task placement constraints. Figure 6 shows the distribution of constraints for all the jobs in the Google trace. Every job would have at least one to a maximum of six unique constraints. A constraint is usually accompanied with one of the three comparison operators (<,>,=) as described in the paper [2] (e.g.,`Kernel_version > INTEGER_value`, `Ethernet_Speed = FLOAT_value`). One can observe from Figure 6 that there are 33% of jobs that ask for two constraints, but only 12% of the worker nodes could satisfy the job's requirements. As the incoming jobs demand more number of constraints, it becomes harder to satisfy all the constraints (chances drop to as low as 5% in case of 6 constraints shown in Figure 6). However, the number of jobs that ask 4 or more constraints is cumulatively about 20%, and the remaining 80% of the jobs had only three or less constraints.

We explore the design space to figure out the optimal probe ratio and heartbeat interval. We find the optimal probe ratio to be 2 as a tradeoff between mis-estimation penalty vs redundant proxy probes being sent to constrained worker queues. We conservatively fix the network delay (a round trip time to synchronize with CRV node monitor) to 0.5 milliseconds, and the cost to calculate CRV ratio is assumed minimal due to the distributed nature of Phoenix refer Section VI-C. To prevent starvation of unconstrained jobs being bypassed by CRV, we compare the performance to constrained jobs in Figure 9. We fix the starvation threshold value to 5, as the number of times a individual task waiting in the worker queue could be bypassed.

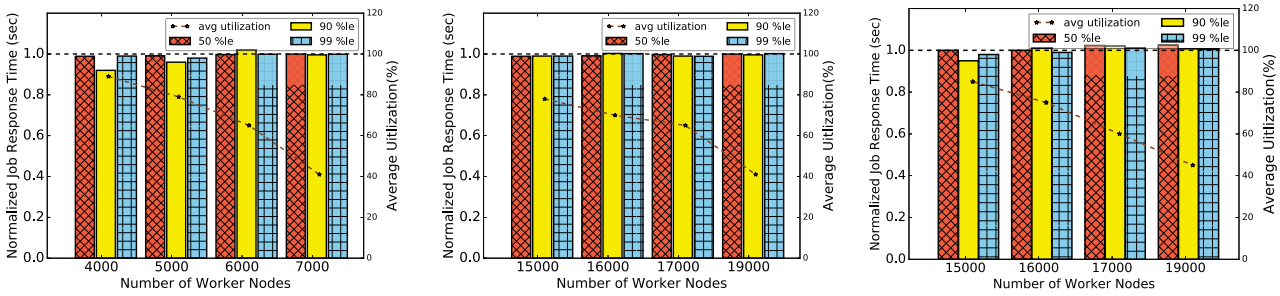
B. Evaluation

We evaluate the proposed scheduler across the three production traces for various cluster utilization loads by varying the number of nodes in the cluster. We focus on moderate to heavily loaded scenarios as the constrained tasks incur more queuing delay due to SRPT. We use metrics [50^{th}, 90^{th}, 99^{th}] percentile of job completion times to compare Phoenix against existing state-of-the-art scheduling schemes like Eagle, Hawk and Sparrow to handle constraints. Since we observed increased queuing delay for constrained jobs during peak congestions as observed from Figures 2a and 2b, it is evident that all the existing schedulers experience tail latencies in case of



(a) Yahoo (b) Cloudera (c) Google

Fig. 7: 50/90/99th percentile response times for short jobs scheduled with Phoenix normalized to Eagle-C scheduler (lower the better)



(a) Yahoo (b) Cloudera (c) Google

Fig. 8: 50/90/99th percentile response times for long jobs scheduled with Phoenix normalized to Eagle-C scheduler (lower the better)

Workload	Nodes	Constrained Tasks	Unconstrained Tasks	Reordered tasks	Short jobs
Yahoo	5000	251404	263240	5227	91.56%
Cloudera	15000	1972428	1925052	817470	95%
Google	15000	6602875	6265616	1717145	90.2%

TABLE III: CRV reordering statistics.

short jobs with constraints. Our focus is toward optimizing for tail latencies of constrained jobs. Since the scheduler is stochastic in selecting workers to send the sample probes to them, we present the results averaged over five runs to ensure consistency.

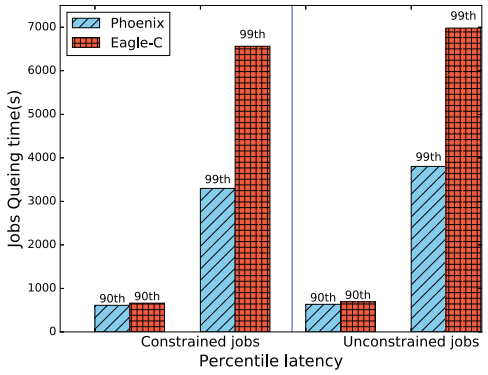


Fig. 9: Comparing queuing delay of jobs in Google trace.

VI. Results and Analysis

A. Comparison of Phoenix to Eagle-C

Figure 7 depicts the improvements in short job response times for varying cluster loads, where the average cluster utilization is shown as the line (aligned to right y-axis). It is observed that, as we increase the number of nodes from 15,000 to 19,000 in the cluster in case of Google trace, the average cluster utilization drops from 86% to 43%. The job response times for all the three traces are normalized w.r.t Eagle-C for varying cluster loads (shown in bars aligned with left y-axis). As can be seen, Phoenix improves by taking only 52% of the job response time of Eagle-C (*1.9*×) consistently across all the traces in high utilizations (with 4000 nodes). As the average cluster load (or utilization) drops, the scheme improvements saturate and converge to the baseline Eagle-C at lower cluster utilizations (last bar). The best improvements of 1.9× come from 85% average utilization in all cases because the CRV based reordering is dynamically invoked only when the constrained queues are congested beyond the `CRV_threshold`, which usually happens in high utilizations. During peak congestions Phoenix does not rely on SBP and instead dynamically estimates the wait time of highly constrained nodes as discussed in Section IV-A. The Phoenix scheduler interacts with the CRV node monitor to reorder the constrained tasks and alleviate the congestion by holistically improving the overall job response times of

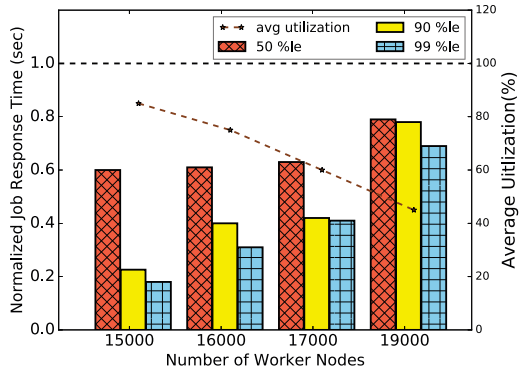


Fig. 10: Phoenix normalized to Hawk-C for Google short jobs.

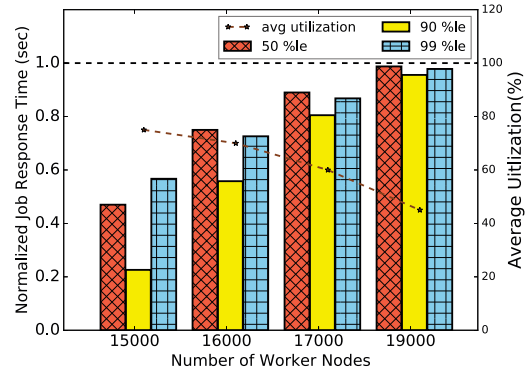


Fig. 11: Phoenix normalized to Sparrow-C for Google short jobs.

constrained jobs. This results in faster job response times along with reduced queuing delays across the constrained queues. The improvements are better for Google traces compared to Yahoo and Cloudera, because Google traces have more variance in the constraint distribution in case of short jobs which results in diverse constraints across individual jobs. Phoenix does not affect the long job response times as shown in the Figure 8, it is noted that CRV based reordering did not affect the job turnaround times of long jobs. Phoenix also avoids starvation of (i) constraint tasks with relatively lower CRV values, and (ii) unconstrained tasks in a worker queue by limiting the number of times a task can be bypassed by other tasks, as implemented in Eagle-C.

B. Phoenix compared with other distributed/hybrid schedulers

We also compare Phoenix to two other distributed and hybrid schedulers namely Sparrow and Hawk.

1) Phoenix compared with Hawk-C: Figure 10 shows job response times of Phoenix normalized to Hawk for Google trace. Job response time at the 90^{th} percentile for Phoenix takes only between 21% (at 86% utilization) to 80% (at 40% utilization) as that of Hawk-C. This translates to a improvement of 4.7× and 1.25× respectively. At 99^{th} percentile Phoenix takes only between 18% (at 86% utilization) to 76% (at 40% utilization) of the job response time of Hawk-C. Again this translates to 5.5× and 1.3× improvement of job response times respectively.

Since Hawk does not do the SBP and SRPT based reordering, it performs very poorly in peak congestions, and this worsens the problem even more in presence of task constraints. Hawk uses random task-stealing in which the probability of task stealing is very low during high loads. In high loads Phoenix estimates the queue waiting times and reorders tasks in those queues, which in turn leads to faster job response times.

2) Phoenix compared with Sparrow-C: As Sparrow is distributed, it is completely agnostic of task runtimes of jobs (does not distinguish between long and short jobs). Hence, the short job turn around times are worsened due to head of the line blocking of long jobs. Thus, Sparrow by itself handles placement constraints in a trivial way. In Sparrow, constrained resources are sampled randomly to place tasks. But they do not handle hard and soft constraints. But Phoenix being a hybrid scheduler, mitigates this problem of starvation of short jobs due to head of the line blocking by dynamically rescheduling the probes of constrained tasks based on CRV.

Figure 11 shows the improvements of job completion times of Phoenix normalized to Sparrow-C for Google trace. Similar to the other schemes Phoenix also outperforms Sparrow by taking only 48% of the job completion times at the 50^{th} percentile with 86% cluster utilization to 95% of the job response times at the 99^{th} percentile with 46% cluster utilization. Note that, the variation between 90^{th} and 99^{th} percentiles are not similar to the other schemes matching the reasons explained above.

C. Performance Overheads of Phoenix

The performance of phoenix is dependent on certain important design choices. First is the frequency of CRV node monitor (heartbeat interval) at which CRV manager synchronizes with worker nodes to calculate CRV ratios. This determines the accuracy of waiting time estimations and round trip time overheads. After a detailed sensitivity analysis which takes the trade-offs between queue waiting time estimation accuracy and round trip time overheads, we empirically set the frequency to 9s. In contrast Yacc+D and Yarn uses 5s as their heartbeat interval for communicating with the node manager. Since we conservatively assume the network communication latency as constant 0.5ms, the network cost to communicate with every worker to initiate reordering becomes negligible. Next, the associated delay to compute CRV is also negligible as it involves trivial logic on simple bit vectors.

D. Breakdown of Benefits of Phoenix

Figure 9 shows the job queuing delay for short jobs in two scenarios: (i) 90^{th} and 99^{th} percentiles of both Phoenix and Eagle-C constrained jobs, and (ii) 90^{th} and 99^{th} percentiles of both Phoenix and Eagle-C unconstrained jobs. We use this data to explain how a constraint-aware algorithm such as CRV can improve the overall queuing delay for *both constrained and unconstrained jobs*: It is evident that Phoenix significantly improves the 99^{th} percentile latency for both the constrained

and unconstrained jobs. In Eagle-C, the constrained jobs experience longer queuing delays due to poorer management of constrained resources and in turn, these jobs also stall the execution of other unconstrained tasks in the same queue. Note that, in such a scenario SRPT reordering would further delay the queuing delay of tasks as it is agnostic of the constrained and unconstrained jobs.

Phoenix on the other hand uses the intelligent CRV based reordering which identifies the constrained jobs and reorder them to execute faster, during these times of peak congestion. This aids in overall speedup for both type of jobs while ensuring the overall fairness of other tasks in the queue.

In summary, we evaluate Phoenix against contemporary distributed and hybrid schedulers like Sparrow, Hawk-C and Eagle-C to find that the tail latency of jobs improves by 1.72×, 4× and 1.9× when compared to jobs without constraints. It improves the average job queuing delays by 1.9× when compared to the state-of-the-art Eagle-C scheduler. It also improves the overall job response times by 2× and 1.6× when compared against Sparrow-C and Hawk-C respectively.

VII. Related Work

In this section, we summarize the features of state-of-the-art schedulers shown in the Table I by broadly classifying these schedulers into three categories based on their design.

A. Centralized and Hierarchical Schedulers

First generation schedulers were fully centralized like the Hadoop capacity scheduler [29] that did slots-based resource management, which lead to poor resource utilizations as the slots-based model suffered from resource fragmentation and did not scale very well in terms of scheduling throughput [15] to meet the sub-second job response times SLAs. To address these issues, Mesos [13] and Omega [21] were designed to be hierarchical. Though Mesos handles placements constraints it does not expose the heterogeneity of the datacenter through hard and soft constraints. All docker based resource managers like Mesosphere [25], Kubernetes [38], and Nomad [39] neither handle task level constraint nor does dynamic task reordering based on resource demand surge.

B. Fully Distributed Schedulers

Distributed schedulers like Sparrow [15] handle placement constraints naively. Sparrow randomly samples from the constrained resource and schedules pending tasks to such sampled queues. But the scheduler does not support task reordering and placement preference constraints. Apollo [40] supports only capacity constraints but does not support task placement constraints. In contrast to Sparrow, Apollo uses much complex wait time estimation based mechanism to minimize the job scheduling delay when compared to our light weight CRV based estimations.

C. Hybrid Schedulers

Mercury [41] is a heterogeneity aware hybrid design. It uses containers that could be queued to schedule short tasks. Mercury does not optimize for dynamic surges in resource demands since it does not handle constraints inherently. Hawk [16] and Eagle [17] line of schedulers are very close to our design in terms of scheduler scalability, but they do not adapt and optimize for job turn around times of constrained jobs.

D. Constraint-aware Schedulers

Most of the related schedulers that are constraint-aware are extended on top of resource negotiators like Yarn which is centralized or two-level like Mesos. Neither of the design choices is optimal for latency critical short jobs. For example, Choosy [20] is a max-min based fairness scheduling in which hard constraints are handled. It is an extension on top of Mesos and hence it suffers from the same scalability and early binding issues while incorporating soft constraints for latency-sensitive tasks. On the other hand, task share fairness [42] handles constraints on heterogeneous workloads. But it prioritizes user level fairness metric in sharing constrained resources instead of the job turn around times as a fairness metric. Tetrisched [23] handles hard placement constraints using a complex constraint definition language called space-time request language (STRL) to do plan ahead reservations in Yarn. Constraint definition language for short lived latency sensitive tasks is a overkill and has a learning curve for end users.

VIII. Conclusion

In this paper, We identify the shortcomings of the existing hybrid and distributed schedulers like Eagle and Sparrow, specifically focus in improving tail latency for tasks with constraints. Motivated by our observations, we propose *Phoenix*, a hybrid constraint-aware scheduler, which uses a centralized scheduler for long jobs and distributed schedulers for short jobs. We analyzed publicly released Google traces to characterize different types of constraints and used a synthesizing model to augment Cloudera and Yahoo traces. We proposed a Constraint Resource Vector (CRV) metric to represent the resource supply/demand based task reordering to improve overall job sojourn times across the distributed task queues. We evaluate Phoenix against Eagle-C with production workload traces from Yahoo, Cloudera and Google, and demonstrate improvements in 99^{th} percentile job response time by 1.9×. Phoenix when compared to Hawk-C and Sparrow-C with Google traces improves the tail latency by 5.5× and 2×, respectively. For mixed workloads with unconstrained and constraint jobs, Phoenix also improves unconstrained job response times by 2×. Further, the CRV based reordering does not affect the long job response times along with ensuring the fairness of the other unconstrained tasks.

Acknowledgments

We are indebted to Ashutosh Pattnaik, Anup Sarma, Haibo Zhang, Iyswarya Narayanan, Prasanna Rengasamy and Xulong Tang for insightful comments on several drafts of this paper. Also, this research is generously supported by NSF grants 1317560, 1320478, 1439021, 1629915, 1629129, 1439057, 1626251.

REFERENCES

[1] K. Rafique, A. W. Tareen, M. Saeed, J. Wu, and S. S. Qureshi, "Cloud computing economics opportunities and challenges," in *Broadband Network and Multimedia Technology (IC-BNMT), 2011 4th IEEE International Conference on*. IEEE, 2011, pp. 401–406.

[2] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 3.

[3] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 2016, pp. 31–44.

[4] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das, "Re-nuca: A practical nuca architecture for reram based last-level caches," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 576–585.

[5] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra, "Improving bank-level parallelism for irregular applications," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016*. IEEE, 2016, pp. 1–12.

[6] Z. A. C. M. T. K. Jagadish B. Kotra, Narges Shahidi, "Hardware-software co-design to mitigate dram refresh overheads:a case for refresh-aware process scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[7] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Vip: virtualizing ip chains on handheld platforms," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 655–667.

[8] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.

[9] A. Martin, D. Jamsek, and K. Agarawal, "Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine," *ICCAD Special Session C*, vol. 7, 2013.

[10] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 57–60, Jul. 2014. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2013.17

[11] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[14] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[15] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[16] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 499–510.

[17] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016.

[18] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 36.

[19] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proceedings of the third ACM Symposium on Cloud Computing*. ACM, 2012, p. 25.

[20] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 365–378.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[22] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[23] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 35.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[25] "Google Mesosphere," https://github.com/mesosphere, 2013.

[26] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[27] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[28] 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 *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10.

[29] "Hadoop Capacity Scheduler," https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2009.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, 2011.

[31] M. HarcholBalter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[32] M. Harchol-Balter, "Queueing disciplines," *Wiley Encyclopedia of Operations Research and Management Science*, 2009.

[33] N. Bansal and M. Harchol-Balter, *Analysis of SRPT scheduling: Investigating unfairness*. ACM, 2001, vol. 29, no. 1.

[34] "Eagle," https://github.com/epfl-labos/eagle, 2013.

[35] "Sparrow ," https://github.com/radlab/sparrow, 2016.

[36] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[37] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems*. IEEE, 2011, pp. 390–399.

[38] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[39] "Nomad," https://www.hashicorp.com/blog/nomad.html, 2013.

[40] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 285–300.

[41] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 485–497.

[42] W. Wang, B. Li, B. Liang, and J. Li, "Towards multi-resource fair allocation with placement constraints," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. ACM, 2016, pp. 415–416.