IoT

MDPI

Article

Constraint-Aware Federated Scheduling for Data Center Workloads

Meghana Thiyyakat [1,*], Subramaniam Kalambur [1] and Dinkar Sitaram [2]

[1] Department of Computer Science and Engineering, PES University, Bangalore 560093, India
[2] Cloud Computing Innovation Council of India, Bangalore 560093, India
* Correspondence: meghanathiyyakat@pesu.pes.edu

Abstract: The use of data centers is ubiquitous, as they support multiple technologies across domains for storing, processing, and disseminating data. IoT applications utilize both cloud data centers and edge data centers based on the nature of the workload. Due to the stringent latency requirements of IoT applications, the workloads are run on hardware accelerators such as FPGAs and GPUs for faster execution. The introduction of such hardware alongside existing variations in the hardware and software configurations of the machines in the data center, increases the heterogeneity of the infrastructure. Optimal job performance necessitates the satisfaction of task placement constraints. This is accomplished through constraint-aware scheduling, where tasks are scheduled on worker nodes with appropriate machine configurations. The presence of placement constraints limits the number of suitable resources available to run a task, leading to queuing delays. As federated schedulers have gained prominence for their speed and scalability, we assess the performance of two such schedulers, Megha and Pigeon, within a constraint-aware context. We extend our previous work on Megha by comparing its performance with a constraint-aware version of the state-of-the-art federated scheduler Pigeon, *PigeonC*. The results of our experiments with synthetic and real-world cluster traces show that Megha reduces the 99th percentile of job response time delays by a factor of 10 when compared to PigeonC. We also describe enhancements made to Megha's architecture to improve its scheduling efficiency.

Keywords: job scheduling; data centers; federated architecture; placement constraints

Citation: Thiyyakat, M.; Kalambur, S.; Sitaram, D. Constraint-Aware Federated Scheduling for Data Center Workloads. *IoT* **2023**, *4*, 534–557. https://doi.org/10.3390/iot4040023

Academic Editors: Arun Ravindran and Reshmi Mitra

Received: 22 September 2023
Revised: 2 November 2023
Accepted: 6 November 2023
Published: 8 November 2023

1. Introduction

In the last decade, extensive research has focused on the usage of IoT in various domains, including healthcare [1], smart transportation [2], and agriculture [3,4]. These applications generate a vast amount of data through deployed sensors, necessitating efficient and cost-effective solutions for data storage, processing, and distribution. Cloud computing and edge computing have emerged as the most popular solutions to support IoT applications. Both of these computing paradigms require an array of networking, storage, and computing components, which are encapsulated in a data center (DC).

IoT applications often have strict latency requirements. For example, automatic driving needs information every 10 ms, with a maximum delay of 10 ms [5]. One way to reduce latency in IoT workloads is to execute parts of a transaction at a fog server instead of the cloud. Fog servers are closer to IoT devices, so communication overhead is lower [6]. This is often performed in applications like autonomous cars, eHealth services, and smart cities [7]. Another way to reduce job response time in IoT workloads is to use hardware accelerators like FPGAs and GPUs. These accelerators can process data much faster than general-purpose CPUs [5,8]. Integrating such devices into the DC increases the heterogeneity of the DC's infrastructure. Given that worker nodes in DCs used by IoT applications are heterogeneous [9], with varying software and hardware configurations, the DC's scheduler must match tasks in the workloads to suitable resources, taking into

consideration these configuration parameters. However, this makes the scheduling process more complex [10], increasing queuing delays by a factor of 2 to 6 [11].

The scheduler encapsulates a task's requirements as task placement constraints. A study by Google [11] revealed that over 50% of the tasks in Google's workload had task placement constraints related to hardware and software machine properties. We use the placement constraint distribution and machine statistics published by the study to enrich existing cluster traces and evaluate the performance of federated scheduling architectures when handling workloads with constraints.

The state of the art in constraint-aware scheduling has been limited to hybrid schedulers [12]. Hybrid schedulers employ a set of probe-based distributed schedulers for short jobs and a single centralized scheduler for long jobs. This enables them to be scalable and fast under low loads. However, due to a lack of coordination between the various scheduling components, and the use of probe-based sampling to place jobs, they perform poorly under high loads as pointed out in previous work [13]. Recent research in the area of scheduling has highlighted the benefits of employing federated architectures [13,14] over hybrid schedulers for achieving fast and scalable scheduling. To the best of our knowledge, the use of federated schedulers in scheduling data center workloads, such as IoT and Big Data workloads, with task placement constraints remains unexplored. The objective of this paper is to address this research gap by studying the performance of federated schedulers in the constraint-aware context.

Federated scheduling architectures split the DC into multiple autonomously managed clusters. A top-level scheduler acts as a load balancer and directs tasks to one of the clusters based on some heuristic. Thus, the scheduling load is distributed across these clusters to achieve speed and scalability. Megha, introduced in our previous work [15,16], is a scalable federated scheduler that uses eventual consistency to make fast scheduling decisions with low scheduling overheads. Pigeon [13] and Hydra [14] are the state of the art in federated scheduling. Hydra is Microsoft's proprietary software and is not available to us for performance comparison. We therefore limit our comparisons to Megha and Pigeon. In our prior work on federated scheduling [15], we compared the performance of Megha with Pigeon with the use of cluster traces. However, the tasks in the cluster traces did not have any task placement constraints, and the comparison was made with the assumption that the resources were homogeneous. In our current work, we implement a constraint-aware version of Pigeon, known as *PigeonC*. We compare PigeonC with Megha using cluster traces enriched with task placement constraints. We also do away with the homogeneity assumption by assigning machine constraints to resources.

In this paper, we assess the performance of federated schedulers for workloads with constraints. We use publicly available cluster traces and synthetically generated traces, augmented with placement constraints, to study the performance of the two architectures. The contributions of the paper are:

1. A modified variant of the Pigeon scheduler, named PigeonC. This extension involves fundamental changes to the design to enable Pigeon's architecture to address task placement constraints;
2. Numerous architectural improvements to Megha's original architecture. The enhancements are geared towards improving job response time and scheduling efficiency;
3. A comprehensive explanation of how placement constraints and machine constraints can be represented, and the data structures used to store them;
4. Two constraint-matching approaches supported by a rationale for their suitability within the context of Megha and PigeonC;
5. A performance comparison of Megha and PigeonC using synthetic and real-world cluster traces. Our results demonstrate that Megha significantly outperforms PigeonC, achieving a minimum tenfold reduction in the 99th percentile delays in job response time;
6. A recommendation of scheduling architecture to be used for IoT workloads and the constraint-matching approach suitable for each architecture.

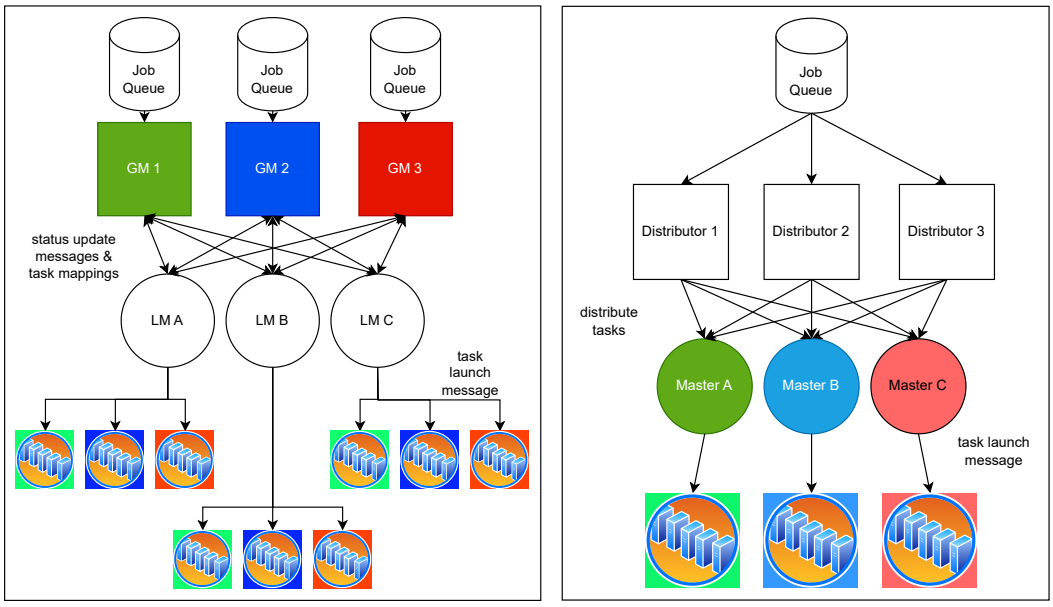
The rest of the paper is organized as follows: Section 2 briefly describes the architectures of Megha and Pigeon, as well as the algorithm used to augment the traces with placement constraints. In Section 3, we give a brief overview of related work in the field of DC scheduling. In Section 4, we delve into the constraint representation matching algorithms. Section 5 offers a detailed overview of the experimental setup, including the cluster traces used, and configuration parameters. Section 6 presents the results of our experiments. We discuss the trends observed in the results and their implications in Section 7. We finally summarize the findings of the paper in Section 8.

2. Background

In this section, we give an overview of the scheduling architectures of Pigeon and Megha. We also briefly summarize the process used to create workloads with constraints.

2.1. Megha's Architecture

Megha [15,16] is a fast, scalable DC scheduling architecture that relies on eventual consistency to provide its multiple scheduling entities with a low overhead and global view of DC resource availability. The two key components of its architecture are Global Managers (GMs) and Local Managers (LMs). Like other federated scheduling architectures [13,14], Megha divides the DC into multiple logical clusters, each managed by an LM. A representation of Megha's architecture is shown in Figure 1a. Each cluster is further divided into sub-clusters. The LM serves as a centralized scheduler and maintains complete and up-to-date information on resource availability within its cluster. It is also responsible for handling failures in both tasks and nodes, as well as sharing the current state of its cluster with all the GMs.



(**a**) Megha's architecture (**b**) Pigeon's architecture

Figure 1. A representation of the architectures of (**a**) Megha and (**b**) Pigeon. In (**a**), GM stands for Global Manager, and LM stands for Local Manager.

On the other hand, the GM is solely responsible for matching tasks with the required resources. Each GM is assigned one sub-cluster, referred to as the GM's 'internal partition', under each LM. The other partitions in the LM's cluster are known as 'external partitions'. In Figure 1a, a GM and its internal partitions are shown in the same color palette.

GMs gather resource availability information from various LMs and aggregate it to create a global view of resource availability in the DC. However, GMs are not immediately informed of changes in an LM's cluster. They learn about these changes through periodic

heartbeats and other forms of communication. This delay can result in a situation where an LM's cluster's resource availability changes, but the GM is not promptly made aware of this change. In such cases, the GM may make invalid scheduling decisions by assigning a task to unavailable resources.

To prevent tasks from queuing at the wrong resources chosen by the GM, Megha employs a validation step. When a GM establishes a mapping between tasks and resources, it sends this mapping to the LM responsible for the resource for validation. If the LM identifies the mapping as invalid, indicating that the requested resource is unavailable, it responds to the GM with a message to this effect. Additionally, the LM also piggybacks the current resource availability information of its cluster on this message. Subsequently, the concerned GM updates its resource availability information and retries scheduling the task. GMs always start the search for resources in their internal partitions before checking external partitions. This ordering reduces the number of invalid requests, particularly under moderate load, and diminishes the likelihood of multiple GMs scheduling tasks on the same resource.

With multiple GMs and LMs operating in parallel, this architecture is fast and scalable, accommodating up to a hundred thousand nodes. Access to global resource availability information reduces unnecessary queuing delays, leading to increased resource utilization and shorter job response times.

2.2. Pigeon's Architecture

Pigeon [13], similarly to Megha and Hydra [14], organizes the data center's resources into smaller logical clusters. This architecture consists of two primary components: Distributors and Masters. Distributors are responsible for evenly distributing a job's tasks among the Masters. Masters, which are analogous to Megha's LMs, each manage a logical cluster or group. The Distributors, Masters, and their respective clusters are shown in Figure 1b. In the figure, the Masters and their clusters share the same colors.

Unlike Megha, in Pigeon, it is the Master, rather than the top-level Distributor, that creates task-to-resource mappings. When a Master receives a task for scheduling, it seeks suitable resources within its cluster for task allocation. If no resources are available, the Master queues the task in its task queue and waits for resources to become free.

Pigeon handles short and long jobs differently. It uses worker reservation for short jobs and employs weighted fair queuing to select tasks from the Masters' queues.

With multiple Distributors and Masters working in parallel, Pigeon achieves low job response time delays at scale. However, since tasks are limited to running on resources in their Master's cluster, available resources in a different cluster cannot be utilized by the task. This results in unnecessary queuing delays and reduced resource utilization. In contrast, Megha does not restrict tasks to particular clusters. When resources are available in the data center, GMs use global resource availability information to schedule pending tasks on resources, regardless of the cluster they belong to.

In our prior work [15], we compared the performance of workloads without constraints across Megha, Sparrow [17], Eagle [18], and Pigeon. Megha, benefiting from its access to global availability information and the flexibility to schedule tasks anywhere in the DC, reported shorter median and average delays compared to the other three schedulers. It reduced average delays for jobs in the Yahoo workload by factors of 1.25, 2, and 1.35 compared to Sparrow, Eagle, and Pigeon, respectively. For jobs in the Google workload, Megha achieved even more significant reductions in average delays, with factors of 12.89, 1.52, and 1.7 compared to Sparrow, Eagle, and Pigeon, respectively.

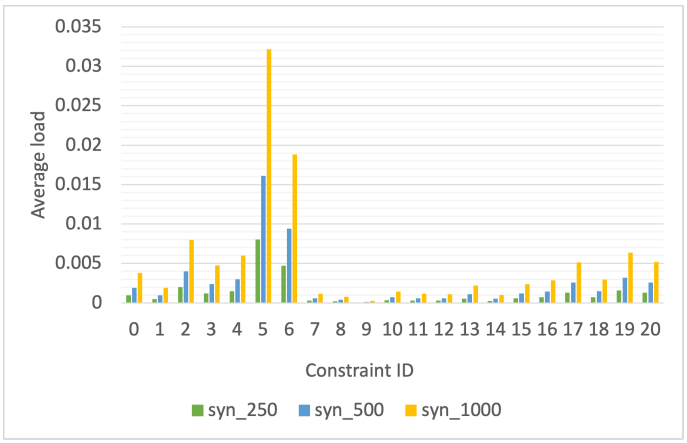
2.3. Real-World and Synthetic Cluster Workloads

In this paper, we utilize both real-world and synthetic cluster traces to generate workloads. Most publicly available cluster traces lack sufficient information about task placement constraints or machine constraints. To address this, we rely on constraint distributions published by Sharma et al. [11], which were synthesized from raw cluster traces

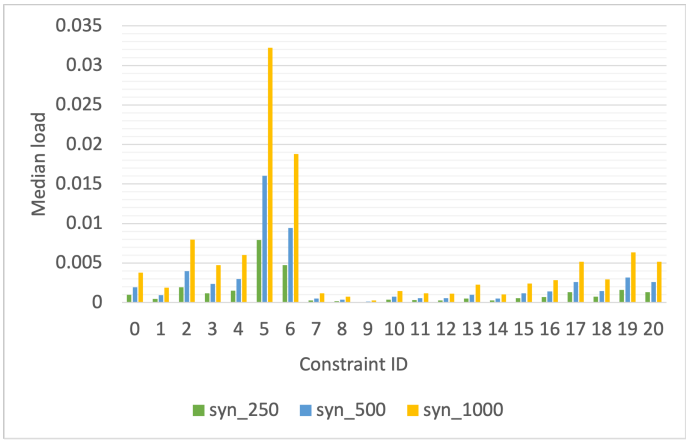
generated from Google's compute clusters. These distributions contain 21 task placement constraints and corresponding machine constraints. These 21 placement constraints represent various software and hardware configuration requirements. For instance, constraint c.1.1 (represented as constraint 0 in our work) signifies a task's requirement for a specific machine architecture, although specific architecture details are obfuscated in the trace. When a machine is assigned constraint 0, it indicates that the machine meets the required architecture type, satisfying the constraint. If a task is assigned task placement constraint 0, it signifies the task's need to be scheduled on a machine with constraint 0, indicating a specific architecture type. Other constraints encode software configurations such as platform family and hardware requirements like the minimum number of disks, cores, and Ethernet speed.

The tasks in the workloads, after augmenting them with constraints, carry a combination of these 21 unique placement constraints. Consequently, a task may have none, one, or more constraints. Each constraint is represented by a unique numerical constraint ID ranging from 0 to 20. The DC's resource load for each constraint in the synthetic workload is depicted in Figure 2. We have labeled the synthetic workloads and traces in the format syn_<number>, where <number> denotes the number of tasks, each lasting 1 s, arriving every second. The resource load for the workload derived from the Yahoo cluster trace [19] is displayed in Figure 3, and the load for the workload derived from a sub-trace of the Google cluster trace [20] is shown in Figure 4. The load for each constraint is calculated, per second, as the ratio of the number of resources required by active tasks needing the placement constraint to the number of resources in the DC that meet the constraint. Because the placement constraints assigned to a task are generated from a probability distribution, the aggregate placement constraints for all active tasks, and therefore the load per constraint ID, vary from second to second. To capture this variability, we have plotted the average, median, 95th percentile, and 99th percentile of the load. In these figures, the x-axis indicates the constraint ID for which the load is calculated.

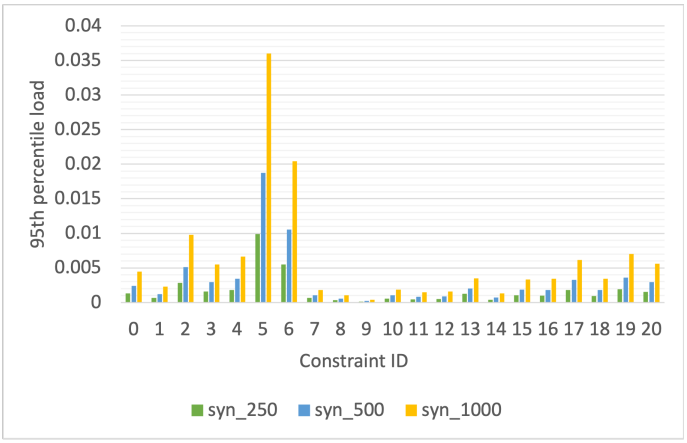
When calculated per constraint in this manner, the load may appear deceptively low. This is because the plots assume that for each constraint, there are dedicated resources that satisfy that constraint and no resources are needed by tasks with other constraints. However, in reality, machine and task placement constraints are a combination of multiple constraints; therefore, a resource satisfying one constraint may also fulfill others. As a result, multiple tasks may require the same worker node. For example, a machine that meets the requirement of one task for a minimum of 4 cores might also meet another task's requirement for a minimum number of disks. Consequently, the same worker node could satisfy the constraints of multiple tasks, leading to potential waiting times for those tasks to access the same worker node. The number of such conflicts, where multiple tasks wait for a worker node to become available, impacts queuing times and, therefore, the delay in job response times. The number of such conflicts encountered by the scheduler each second while scheduling the synthetic workloads is shown in Figure 5. It is important to note that these calculations make two assumptions: first, that tasks have the flexibility to switch to another worker node when needed, and second, that tasks do not experience any delays that might cause their execution to extend into the next few seconds. This scenario represents an idealized view of scheduling complexity and provides insight into the reasons behind higher job response time delays in workloads with constraints.



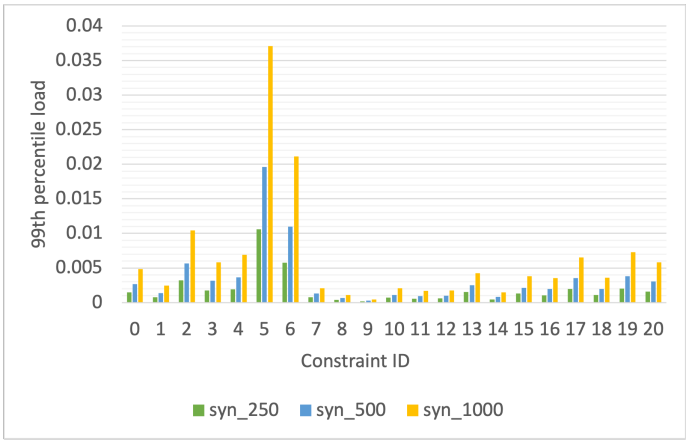
(**a**) Average per-constraint load



(**b**) Median per-constraint load



(**c**) 95th percentile per-constraint load



(**d**) 99th percentile per-constraint load

Figure 2. Per-constraint load applied on the DC by the synthetic trace workloads. syn_250, syn_500, and syn_1000 are the synthetic workloads that have 250, 500 and 1000 tasks, of 1s duration, arriving each second, respectively.

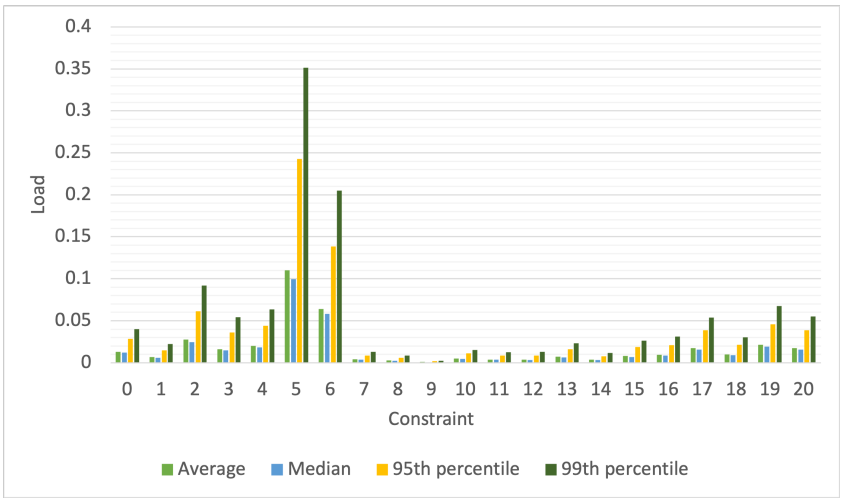


Figure 3. Per-constraint load of the Yahoo workload.

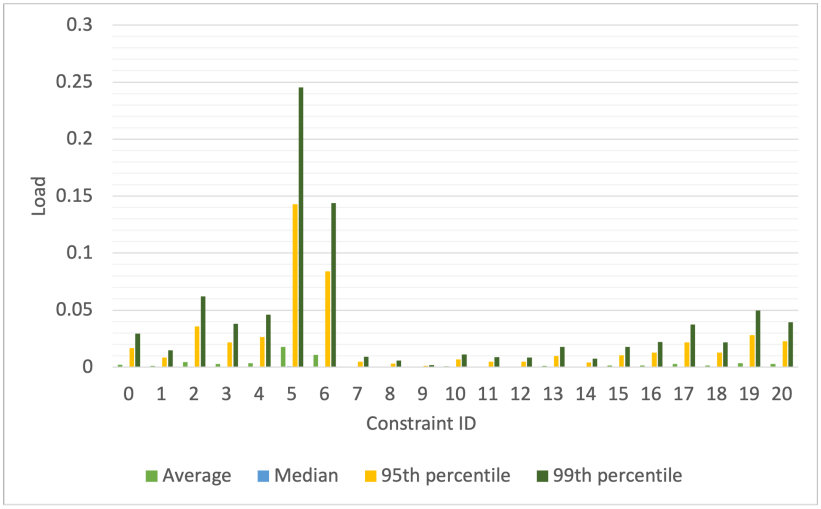


Figure 4. Per-constraint load of the Google workload.

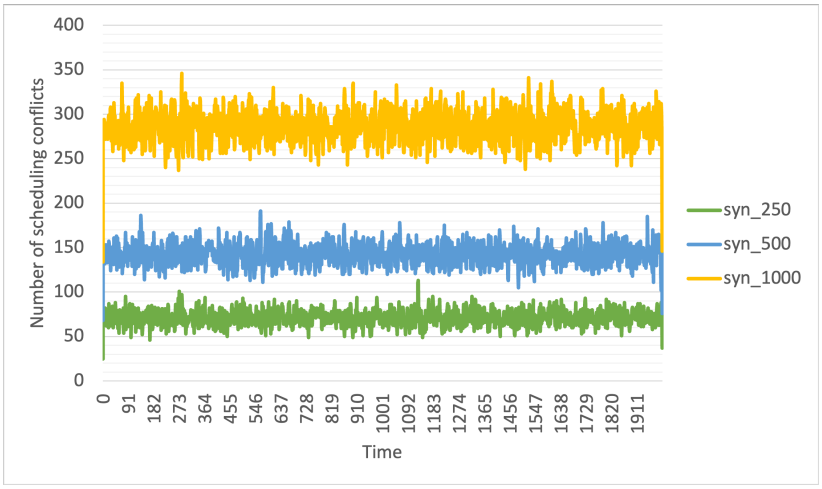


Figure 5. Scheduling conflicts encountered while scheduling the 3 synthetic workloads: syn_250, syn_500, and syn_1000. number in syn_<number> indicates the number of tasks arriving every second.

Another factor that impacts queuing times, regardless of placement constraints, is the overall load on the resources of the DC. This load is calculated as the ratio of the total number of resources required to the total number of resources present in the DC, calculated per second. For the synthetic workloads and the real-world workloads, the median, average, and 99th percentile loads are shown in Table 1. The Yahoo workload exhibits the highest load, with the 99th percentile exceeding 1. This indicates that, during the execution of the workload, the demand for resources outweighs the supply, leading to queuing. In such cases, queuing is inevitable. The other workloads, in the decreasing order of 99th percentile load, are syn_1000, Google, syn_500, and syn_250.

Table 1. The median, average, and 99th percentile load of the workloads generated from the cluster traces.

Cluster Trace	Median Load	Average Load	99th Percentile Load
Yahoo	0.30	0.33	1.08
Google	0.0032	0.05	0.77
syn_250	0.025	0.025	0.025
syn_500	0.05	0.05	0.05
syn_1000	0.1	0.1	0.1

2.4. Metrics

Job response time (JRT) is a widely used metric for comparing the performance of different scheduling frameworks [12,13,18,21]. In our work, we use the delay in job response time to compare scheduler performance. The delay is calculated as follows:

$$delay = ActualJRT - IdealJRT \quad (1)$$

Here, $delay$ represents the delay in job response time, and $IdealJRT$ is the time taken from the submission of a job at the scheduler to its completion in the absence of delays. $ActualJRT$ is the job response time (the time between submission and completion of a job) including the delays encountered, such as execution delay, queuing delay, processing delay, and communication delay. Most of these delay components are inherent to the scheduling framework, making the delay in job response time a valuable metric for assessing scheduler performance in a workload-agnostic manner. Lower delays not only signify improved task performance, but also indicate higher resource utilization. This is because, when there is a scope for delays to be reduced, it suggests that idle resources are waiting for task assignments. The result of the reduction in queuing delays is fewer idle resources, which means higher resource utilization. Enhanced resource utilization is a crucial metric for data center service providers as it leads to a better return on investment.

In addition to delays in JRT, we also use the utilization of the workloads reported under each scheduler to compare the scheduling efficiency and analyze the cause of the delays in JRT. $Utilization\%$ for a workload, at a time t is calculated as:

$$Utilization\%(t) = \frac{R_{consumed}(t)}{R_{total}} \quad (2)$$

Here, $Utilization\%(t)$ represents the resource utilization of the workload at time t, $R_{consumed}$ denotes the number of resource units consumed by all active tasks at time t, and R_{total} is the total number of resource units the DC contains.

3. Related Works

In this section, we give a brief overview of existing work in the areas of DC scheduling, and constraint-aware scheduling.

3.1. Data Center Scheduling

Traditionally, schedulers were created with a centralized entity with global resource availability information [22,23]. However, the centralized design led to the formation of bottlenecks, limiting the scalability and throughput of the scheduler [14]. To solve this problem, schedulers [17,24] with greater degrees of parallelism, which used random probing to achieve high throughput task placements, were developed. These suffered from the drawback of unnecessary worker-side queuing under high loads as a result of the parallel scheduling entities working with partial resource availability information. Hybrid schedulers such as Hawk [21] and Eagle [18] tried to combine the benefits of centralized and distributed schedulers by employing both kinds in their architecture. While this allowed the scheduler to handle short and long jobs differently, thereby catering to their specific performance requirements of low latency and high performance, respectively, the lack of coordination between the two types of schedulers, as well as among the distributed schedulers, resulted in poor performance at high loads. To address the increasing requirements for speed and scalability, federated schedulers gained popularity [13,14]. These schedulers divided the scheduling responsibility across multiple autonomous scheduling entities, which were capable of making independent scheduling decisions without much coordination. This made the architectures scalable, tolerant to failures, and fast. Megha and Pigeon are the two federated schedulers studied in this work. Both divide the data center into multiple smaller clusters, each of which is governed by an autonomous scheduling entity. The top-level scheduling entities send task requests to the bottom-level scheduling

entities to be fulfilled by the resources in their cluster. This distribution of responsibility between the two tiers, and the parallelism present in both tiers makes both federated schedulers scalable and fast. In the case of Megha, the top-level entities operate with a global view of resource availability, allowing Megha to find idle resources even under high load.

Hydra [14] is Microsoft's proprietary federated scheduler that can scale to tens of thousands of workers, and delivers scheduling speeds of up to 40,000 scheduling decisions per second. Such high scalability and scheduling throughput are achieved by Hydra by using a federated architecture that logically divides the DC or large cluster into smaller sub-clusters. Each sub-cluster is governed by a Resource Manager (RM), which handles the life cycle of tasks and jobs. The RM collects and aggregates resource availability information from its worker nodes through periodic heartbeats. It also schedules jobs onto resources based on configured policies. Each worker node runs a Node Manager (NM) component that is in charge of the tasks running on that worker node. Since the Resource Manager is adopted from YARN [23], a centralized resource management architecture, it inherits the scalability limitations of YARN ($\sim$4000 worker nodes). In addition to the RM and NM, Hydra contains a single Global Policy Generator that is in charge of deciding the scheduling policy to be used by the different sub-clusters, Routers that distribute jobs to different sub-clusters, and the AM–RM Proxy. The AM–RM Proxy runs on each worker node and allows jobs to span multiple sub-clusters by posing as the application manager (AM) requesting resources from different RMs, and also acting as an RM when a job's AM requests for resources. Unlike Megha and Pigeon, the federated cluster managers in Hydra communicate with one another to coordinate the placement of tasks. Additionally, the top-level component, the GPG, unlike the GM or the Distributor, is not actively involved in the scheduling of each individual job or task. Since Hydra is Microsoft's proprietary software, we were unable to compare the performance of Hydra with Megha and PigeonC in a constraint-aware setting.

3.2. Constraint-Aware Scheduling

Cloud computing has been used extensively across different domains. This has resulted in heterogeneous DCs with specialized hardware and software, as well as heterogeneous workloads [25]. For a scheduler to make the most of its infrastructure and guarantee the required job performance, it is crucial that the scheduler be aware of the heterogeneity of the DC's resources [26]. Nevertheless, workloads with constraints introduce unique challenges [27]. They can result in an increase in job delays by a factor of 2 to 6 [11].

Phoenix [12] is a constraint-aware, hybrid scheduling architecture, which employs a centralized scheduler for long jobs and distributed schedulers for short jobs. Phoenix periodically calculates the ratio of the demand and supply for each constraint on each worker. It then uses these values to estimate the wait times of the tasks queued at each worker. When the wait time of a worker queue crosses a threshold, Phoenix reorders the tasks in the queue based on the ratio calculated earlier. Phoenix improves the 99th percentile response times of jobs by a factor of 1.25 and 1.9 when compared to Hawk-C and Eagle-C, respectively. Hawk-C and Eagle-C are constraint-aware versions of hybrid schedulers Hawk [21] and Eagle [18] implemented by the authors for comparison. The scheduling process followed by the constraint-aware versions of the hybrid schedulers has not been described by the authors. As mentioned earlier, hybrid schedulers suffer from the limitation of being unable to find idle worker nodes at high loads.

To the best of our knowledge, incorporating constraint-awareness in scheduling architectures halted at hybrid architectures. Hierarchical/federated schedulers such as Hydra and Pigeon have not been evaluated for constraint-awareness, even though they have proved to perform better than hybrid schedulers. In this paper, we evaluate Megha and PigeonC, two constraint-aware federated schedulers, with real-world and synthetic workloads. Neither PigeonC nor Megha employs worker-side queuing; they send tasks to idle worker nodes only. In hybrid approaches such as Phoenix, tasks are queued at worker

nodes based on heuristics, such as task runtime estimates or worker queue lengths, to determine at which workers a task is likely to incur lower queuing delays. However, such estimates cannot accurately determine the worker node at which the task is guaranteed to experience the smallest queuing delay. Moreover, while the scheduling entities in Phoenix also aggregate and periodically update a local copy of the global resource availability information to make scheduling decisions, unlike Megha, they do not employ a validation step that prevents tasks from being queued at busy workers as a result of stale state stored in the scheduling entity. This inconsistency can lead to unnecessary queuing delays. Neither Megha nor Pigeon uses probe-based sampling to find idle workers. Instead, they split the data center into multiple clusters, each of which has a master governing the cluster autonomously. Each master has complete resource availability knowledge of its own cluster. In Pigeon, the master uses this resource availability information to allocate resources to tasks such that there is no worker-side queuing. In Megha, the top-level scheduling entities use global resource availability information to map tasks to resources anywhere in the data center, and the cluster masters (Local Manager) validate the task placements. This approach not only removes the need for worker-side queuing, but the additional flexibility in task placement also reduces queuing at the scheduler and improves resource utilization of the data center.

Medea [28] is a scheduler with an architecture similar to a hybrid architecture. It has one scheduler to schedule long-running applications and a traditional task-based scheduler for other tasks. Medea requires constraints to be captured in the form of tags, which also allows for flexibility in specifying constraints such as affinity, anti-affinity, and cardinality. Neither Megha nor PigeonC considers placement constraints related to co-located tasks. We want to evaluate this aspect of constraint-based scheduling in the future. The long-running application scheduler uses integer linear programming to model the task placement problem with a set of constraints. Medea, however, does not accommodate placement constraints in shorter tasks. The authors suggest that some heuristics be used in the task-based schedulers to cater to such tasks, without overloading the long-running application scheduler.

George [29] is a reinforcement learning-based scheduler specifically designed for long-running applications. George is both interference and constraint-aware. It uses a projection-based proximal policy optimization algorithm along with integer linear optimization techniques to filter the action space. The scheduler also uses transfer learning by leveraging the fact that scheduling events are similar and therefore, the training need not start from scratch and can instead use a base model. This significantly reduced the training time. When compared to Medea, George not only reduces the training time by a factor of 16 but also reduces constraint violations by a factor of 8.5. Medea also reports 23% lower container requests per second. This is attributed to George's awareness of inter-container interference. Megha does not discriminate between tasks of varying durations, while PigeonC treats shorter jobs with higher priority. PigeonC uses weighted fair queuing while choosing tasks to schedule, thereby prioritizing tasks from short jobs and reducing their queuing delay.

Tetrisched [30] is a heterogeneity-aware scheduler, that leverages runtime estimates, generated from information collected by the reservation system, to plan ahead. Task requirements are captured using a declarative language known as Space-Time Request Language. Tetrisched periodically constructs a Mixed Integer Linear Programming formulation to update scheduler plans by considering the system as a whole. Tetrisched has been integrated with the YARN scheduler and has shown higher SLO satisfaction and reduced latency in best-effort jobs when compared to the state-of-the-art YARN scheduling stack.

Swain et al. [31] define a Constraint Aware Profit Maximization problem in terms of the revenue generated from a task, the cost of the resources to the service provider, and the penalty incurred due to missed deadlines. They study the combined effect of task placement constraints, and deadline constraints on profits and try to maximize the latter using a profit-based heuristic approach called the Heuristic of Ordering and Mapping for CAPM problem (HOM-CAPM), which controls task admissions and task allocation. The

approach also involves grouping tasks and machines based on hard constraints, thereby reducing the number of worker nodes that need to be searched for a particular task. This approach is only possible when the hard constraints are static and known to the scheduler beforehand such that mutually exclusive, meaningful groupings can be created. Since Swain et al. use only the Google cluster trace to evaluate their approach, the assumption of knowing the hard constraints holds. In our study, the scheduling approach is agnostic of the type of constraint and therefore can be used without modification for all workload traces even when machines cannot be grouped. Furthermore, Megha and Pigeon do not explicitly consider profit maximization and only focus on reducing delays in job response time. However, this reduction in delays has an impact on the utilization of resources, QoS, and therefore the revenue, profits, and penalty. Additionally, our work also focuses on the underlying architecture of the scheduling approach taking into consideration the speed-up and scalability offered by having multiple scheduling entities working in parallel.

While the authors may not explicitly mention it, the underlying architectures of George, TetriSched, and HOM-CAPM appear to be centralized. Centralized architectures, as discussed earlier, have limitations of both bottleneck formation and low scalability. Megha and PigeonC, on the other hand, have federated architectures involving multiple scheduling components functioning in parallel, enabling fast and scalable scheduling.

4. Constraint-Aware Scheduling with Megha and PigeonC

In this section, we describe how we have incorporated constraint awareness in the existing architectures of Pigeon and Megha.

4.1. Representation

In both PigeonC and Megha, we have used bit vectors to represent the machine constraints and task placement constraints. Let $C = (C1, C2, C3, \ldots, CK)$ be the K machine constraints supported by the DC's infrastructure. When we say a worker node has a machine constraint Ck, $Ck \in C$, we mean that the worker node satisfies the task's hardware or software requirement encoded as Ck. On the other hand, when we say a task has a placement constraint Ck, we mean that the task requires that the hardware/software requirement Ck be met by the worker node; that is, the worker node must have a machine constraint Ck. Therefore, a task T with a set of placement constraints C_T, such that $C_T \subseteq C$, can only be scheduled on a machine W with machine constraints C_W if $C_T \subseteq C_W$; that is, all the task placement constraints of task T are satisfied by or contained in W's set of machine constraints, C_W.

4.1.1. Representation in Megha

Megha divides the DC into smaller logical clusters $(L_1, L_2, \ldots, L_N)$, where N is the number of clusters. Each cluster, L_i, is further divided into partitions $(P_i^1, P_i^2, \ldots, P_i^M)$, where M represents the number of partitions. We use bit vectors to represent the constraints satisfied by the worker nodes in a partition [16]. For a partition P_i^j in cluster i, we represent the constraints satisfied by the nodes in the cluster as a set of bit vectors $B_i^j = (Bk_i^j, B2_i^j, \ldots, BK_i^j)$. If the partition consists of Q workers, $P_i^j = (W1_i^j, W2_i^j, \ldots, WQ_i^j)$, the bit vector for a constraint k, Bk_i^j, is an ordered list of Q bits $[bk_{W1_i^j}, bk_{W2_i^j}, \ldots, bk_{WQ_i^j}]$, where each bit, $bk_{Wl_i^j}$, represents whether or not worker Wl_i^j satisfies constraint k. Therefore, if a partition P_i^j, belonging to cluster L_i contains five worker nodes $W = (W1_i^j, W2_i^j, W3_i^j, W4_i^j, W5_i^j)$, with worker nodes $W1_i^j, W2_i^j, W5_i^j$ satisfying constraint k, then $Bk_i^j = [1, 1, 0, 0, 1]$.

4.1.2. Representation in PigeonC

Like Megha, PigeonC also divides its DC into multiple small logical clusters or groups, $(L_1, L_2, \ldots, L_N)$, where N is the number of clusters. However, the clusters in PigeonC are not divided any further into partitions. Therefore, we represent the constraints sat-

isfied by the nodes in cluster $L_i = (W1_i, W2_i, \ldots, WQ_i)$ as a collection of bit vectors $B_i = (B1_i, B2_i, \ldots, BK_i)$, and each bit vector Bk_i is an ordered list of Q bits, such that $Bk_i = [bk_{W1_i}, bk_{W2_i}, \ldots, bk_{WQ_i}]$ represents which workers in the cluster satisfy constraint k. When a bit bk_{Wl_i}, such that $bk_{Wl_i} \in Bk_i$ is set to 0, it means worker Wl_i does not have/satisfy the machine constraint Ck. Conversely, when bit bk_{Wl_i} is set to 1, it indicates that the worker node Wl_i has/satisfies the constraint Ck.

4.2. Constraint-Awareness in PigeonC

As explained in Section 2.2, Pigeon divides the DC into clusters. The Distributors receive jobs and distribute the tasks in a job evenly to all the Masters. In PigeonC, the constraint-aware version of Pigeon, we modify how tasks are distributed to Masters. All Distributors in PigeonC maintain a data structure, CS_i that represents, for each unique set of machine constraints, the frequency of occurrence of the constraint set in the cluster i. That is, if workers $W1_1$, $W2_1$, $W3_1$ in cluster L_1 have machine constraints $(C1, C3, C5)$, $(C1, C2)$ and $(C4)$, respectively, and if workers $W1_2$, $W2_2$ and $W3_2$ in cluster L_2 have machine constraints $(C2, C4)$,$(C4)$ and $(C2, C4)$, respectively, the constraint set occurrence for L_1 would be:

$$CS_1 = \{(C1, C3, C5) : 1, (C1, C2) : 1, (C4) : 1\}$$

Similarly, the constraint set occurrence for L_2 would be:

$$CS_2 = \{(C2, C4) : 2, (C4) : 1\}$$

When a task T, with task placement constraints C_T, needs to be assigned to a Master the Distributor aggregates the counts of the constraint set occurrences in $CS_i = (cs_A, cs_B, \ldots, cs_S)$ for each cluster L_i, where cs_X is the frequency of occurrence of unique constraint set X and $T_C \subseteq X$. That is, the aggregate count, $count_i^C$ of the constraint set occurrence in cluster L_i, T_C is:

$$count_i^C = \sum_{cs_X \subseteq CS_i,\, X \supseteq T_C} cs_X$$

The Distributor uses the $count_i^C$ as weights, where each weight corresponds to the availability of constraint sets in each cluster, and uses weighted random selection to pick a Master. This ensures that the load is distributed as per the supply of the constraint sets available in each cluster.

Once the task is sent to a Master, the Master uses the matching algorithm described in the following section to choose an appropriate worker node for the task such that the worker node has the required resources to run the task and satisfies the placement constraints of the task.

In addition to the changes mentioned above, we have also removed the reservation of worker nodes for short jobs, which is a feature present in Pigeon. This was performed to address the potential issue of indefinite task starvation for long jobs when suitable resources are only reserved for short jobs. This change ensures better resource utilization and fairness in job scheduling. PigeonC continues to use weighted fair queuing to give priority to short jobs over long jobs, if the fair queue weight is FQW, PigeonC tries to schedule one task from a long job for every FQW number of tasks from short jobs. However, when idle resources in the DC do not match any of the tasks from the short jobs, PigeonC schedules suitable tasks from long jobs onto these resources.

4.3. Matching Algorithm

Since both Megha and PigeonC have a bit vector representation of the machine constraints, similar algorithms can be used to find a suitable worker node for a task T with task placement constraints C_T. Both architectures employ a greedy matching algorithm while choosing suitable resources for tasks. Since real-world workloads are unpredictable in terms of task execution times, task resource demands, and job fan-out degree, finding

an optimal task-to-resource mapping for the set of all the tasks in the workload, such that queuing delays are optimally minimized, is not realistic. Instead, we consider each task individually and try to find the best possible resource match for the task. However, for a particular task, there may be multiple suitable worker nodes. We evaluate two techniques for choosing a worker node from the set of suitable worker nodes: Random, and Minimum Constraints. We formally describe the matching algorithms and the heuristic employed to reduce queuing delays below.

Suppose W is the set of workers in the chosen partition or cluster, in the cases of Megha and PigeonC, respectively, then, $\forall Ci \in C_T$, the corresponding bit-vectors are $\{B_i \mid C_i \in C_T\}$. The matching algorithm used in both PigeonC and Megha is depicted in Algorithm 1. The input to the algorithm consists of the list of bit vectors B, the resource availability vector R, and the task placement constraints required by a task, C_T. The resource availability vector is a vector of 1s and 0s, which indicates whether or not a worker node, represented by a single bit at the corresponding index, is available (1) or busy (0). Bit-wise AND operations are performed on the bit vectors for each task placement constraint in C_T, and the resource availability vector. The time complexity of finding all possible suitable worker nodes in a partition or cluster, in Megha or PigeonC, respectively, can be represented as:

$$\mathcal{O}(match(T)) = \mathcal{O}(n.m) \tag{3}$$

where n is the number of placement constraints in C_T, and m is the number of bits in a bit-vector B_i, such that $Ci \in C_T$. Since all partitions and clusters are of the same size, the number of bits representing each of them in bit-vector B_i will be the same for a given DC configuration. However, since the AND operation between the vectors is likely to be performed at the byte or word level, the time complexity in Equation (3) can be re-written as:

$$\mathcal{O}(match(T)) = \mathcal{O}(n.log_{size}m) \tag{4}$$

where *size* represents the number of bits in the operands of the AND operation performed in the CPU.

Algorithm 1: Matching algorithm.

Data: list of bit vectors B, set of task constraints C_T, resource availability vector R
Result: suitable worker node W
`// Initialize resultant vector` *result*
result $\leftarrow R$;
`// Iterate through` K `bit vectors`
foreach $i \in C_T$ **do**
  result $\leftarrow$ *result* AND $B[i]$;
end
suitable_workers $\leftarrow$ indices of bits set to 1 in result
$W \leftarrow$ worker node chosen from suitable workers

After the bit-wise AND operation is performed, the bits set to 1 in the resultant vector are worker nodes that are suitable for the task. That is, the worker nodes are available to execute the task and have all the machine constraints required to satisfy the task's placement constraints. We have evaluated two approaches, *Random* and *Minimum Constraints*, to choose a worker node from the set of suitable worker nodes.

4.3.1. Random

In this approach, the final worker is selected through uniform random selection. This means that any suitable worker node has an equal probability of being chosen from the set of all suitable worker nodes. Using this approach in Megha offers a notable advantage. The introduction of randomness into worker node selection reduces the likelihood of multiple GMs selecting the same worker node. When multiple GMs attempt to schedule a task on

the same idle worker node, it results in at least one invalid request because an idle worker node can execute only one task at a time. Invalid requests lead to communication and processing overheads. The probability of such invalid requests increases significantly under high loads due to the limited availability of idle worker nodes.

4.3.2. Minimum Constraints

While the Random approach may help reduce inconsistencies in Megha under high loads, the performance advantages of this approach might not be as significant in Megha under low-load conditions or in PigeonC. Workers with a greater number of machine constraints are inherently more versatile, as they can match a wider range of tasks. Opting for the Random approach to choose a suitable worker node might be inefficient, as it could select a worker with a constraint set much larger than the task's constraints. This could reduce the pool of suitable worker nodes available for future tasks and potentially lead to additional queuing delays in task execution. For example, suppose the DC had two idle worker nodes, $workerA$, with machine constraints $(1, 2, 3, 4)$, and another worker node, $workerB$, with machine constraints $(1, 2)$; then, for a task, $task1$, with task placement constraints $(1, 2)$, both worker nodes would be considered as suitable. Nevertheless, $workerA$ supports more constraints than $workerB$. If $workerA$ was chosen at random for $task1$, and another task, $task2$ arrives with task placement constraints (3), $task2$ cannot execute until $task1$ completes because $workerB$ does not satisfy $task2$'s placement constraints. However, had $workerB$ been chosen for $task1$'s execution, then $task2$ would have been scheduled onto $workerA$ without any queuing delay. Therefore, to improve the likelihood of finding suitable worker nodes for future tasks with placement constraints that are a subset of $workerA$'s constraints, we use the Minimum Constraints approach which is similar to the best-fit algorithm. In this approach, we choose the worker node, from the set of suitable worker nodes, with the least number of machine constraints. The approach can be represented as follows:

$$W_{\text{chosen}} = \arg \min_{W \in S_{\text{suitable}}} |machine\_constraints(W)|$$

Here, W_{chosen} is the suitable worker node with the minimum constraints, S_{suitable} is the set of all suitable worker nodes satisfying the task's placement constraints, and $machine\_constraints(W)$ is a function that returns the number of machine constraints supported by a worker W. By employing this approach, we do not hinder the task's performance because we have already determined the chosen worker node as suitable. Simultaneously, we enhance the chances of future tasks finding compatible worker nodes. In the scenario described earlier, $workerB$ has the least number of machine constraints and would have been chosen for $task1$, leaving $workerA$ available to execute $task2$, without $task2$ encountering queuing delays.

4.4. Enhancements Made to Megha

In addition to the matching algorithm, Megha's architecture has undergone a few enhancements to improve the efficiency of the scheduler and to reduce the job response time of the workload. The highest contributor to the processing overhead in Megha's original architecture [15,16] was the periodic and aperiodic status updates from the LMs to the GMs. These updates included a comprehensive snapshot of the resource availability information for the entire cluster and were transmitted to the GMs following inconsistencies or LM heartbeat intervals. In the revised design, LMs no longer send the GMs the entire snapshot; instead, they only transmit the changes made to the cluster's resource availability since the last status update was shared with the GM. Consequently, separate delta updates are maintained for each GM.

In the earlier design, one of the causes for inconsistencies was the arrival of a task completion message indicating that the task has freed resources, closely followed by a status update message also indicating the same. In such cases, the GM might allocate a task to the recently freed resources and, upon receiving a status update message again

indicating their availability, mistakenly assign another task to the same resources. This led to the second request failing in the validation step at the GM resulting in an inconsistency, prompting yet another status update from the LM. To prevent this issue, status updates sent to a GM no longer include information about resources that became available due to the GM's tasks completing their execution. Instead, this information is conveyed to the GM through task completion messages.

4.5. Limitations

In our work, we have assumed that all task placement constraints are hard constraints, meaning that we consider these constraints as requirements rather than preferences (soft constraints). Furthermore, we do not account for task placement constraints that depend on the placement of other tasks. For example, we do not address constraints requiring two tasks to be scheduled on the same machine.

5. Evaluation Setup

In this section, we describe the experimental setup used to compare the performance of Megha and PigeonC.

Previous studies [12] related to constraint-aware scheduling have relied upon trace-driven simulation to study the performance of scheduling architectures. We implemented both Megha and PigeonC on a publicly available trace-driven simulator [32] used in previous works [13,16–18,21] to evaluate scheduling architectures. We used publicly available Yahoo cluster trace [19] and a sub-trace of the Google cluster trace [20] along with synthetically generated traces as input to the simulator. The Yahoo cluster trace consisted of 24,262 jobs and 968,335 tasks. The Google cluster sub-trace consisted of 10,000 jobs and 312,558 tasks. The synthetically generated traces consisted of 2000 jobs arriving with a constant job inter-arrival time of 1 s. Each job was made up of 250/500/1000 tasks that ran for 1 s each. Therefore, the synthetic traces tested the schedulers' performance at constant load. In this paper, the workload generated from the Yahoo cluster trace has been referred to as the Yahoo workload, the workload from the Google cluster sub-trace as the Google workload, and from the synthetic traces, as the syn_250, syn_500, and syn_1000 workloads. The <number> in syn_<number> indicates the number of tasks, of duration 1s, arriving per second. All the workloads were augmented with task placement constraints.

Following the approach of previous research [13,21,32], we assume the tasks in a job are independent and do not have inter-task dependencies. Therefore, the value for $IdealJRT$, from Equation (1), was set to the duration of the longest task. That is, in the ideal scenario with no delays encountered, the job would finish whenever its longest task would.

The DC size for the runs with synthetic cluster traces and the Yahoo cluster trace was set to 10,000 worker nodes. For the Google cluster sub-trace, it was set to 20,000.

Both Megha and PigeonC were built on a publicly available event-driven simulator that takes as input a cluster trace. Each line in the cluster trace represents a job in the form <arrival time, number of tasks, average task duration, duration of task 1, duration of task 2, duration of task 3, . . .>. The simulator was implemented in Python. The reproducibility of the simulation study was maintained as long as the seeds to initialize the random number generator remained unchanged across runs. We used a pseudo-random number generator from Python's library while assigning task placement constraints and machine constraints as per the algorithms published by Sharma et al. [11] and for the Random approach.

For each cluster trace and type of constraint-matching approach, we conducted simulations for both Megha and PigeonC using three different seed values for the random number generator. Furthermore, we employed distinct configurations for each of the three runs, resulting in varying constraint assignments to the machines across configurations. The delays in job response time shown in the next section are the average values of the delays that were reported in the three runs.

Since PigeonC uses weighted fair queuing, we set the value of *FQW* to 20 in all the experiments. This value was chosen based on the results of experiments conducted in a previous work [13], which showed that Pigeon performed best with this value of *FQW*.

6. Results

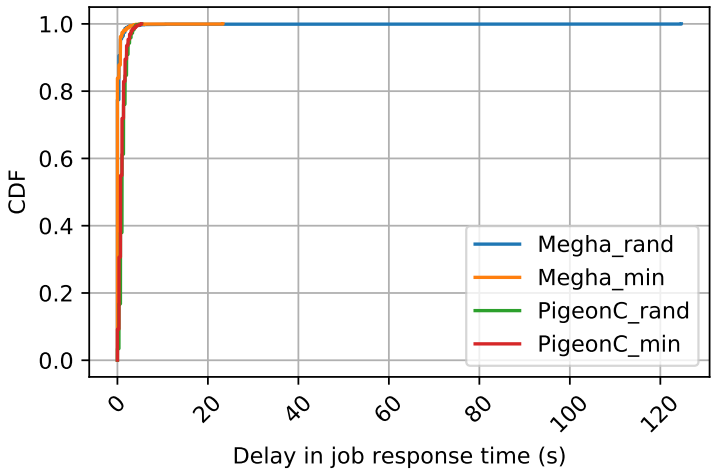
This section reports the results of the experiments conducted with the two schedulers, Megha and PigeonC, using real-world and synthetic workloads.

The CDF of the average delays in job response time of the various jobs in the synthetic workloads, Yahoo workload, and Google workload are shown in Figures 6–8. The delays have been calculated as shown in Equation (1). The figures contain the CDFs of Megha and PigeonC for the two matching algorithms: Random and Minimum Constraints. The graphs for Megha are labeled Megha_rand and Megha_min for the CDFs of the delays under Megha with the Random approach and the Minimum Constraints approach, respectively. Similarly, the graphs for PigeonC are labeled PigeonC_rand and PigeonC_min for the CDFs of the delays under PigeonC with the Random and the Minimum Constraints approach, respectively. In the figures, a graph for a particular scheduler-matching approach combination may not be visible due to an overlap with the graph of the scheduler with the alternative approach, indicating that the performance did not vary significantly under the two approaches.

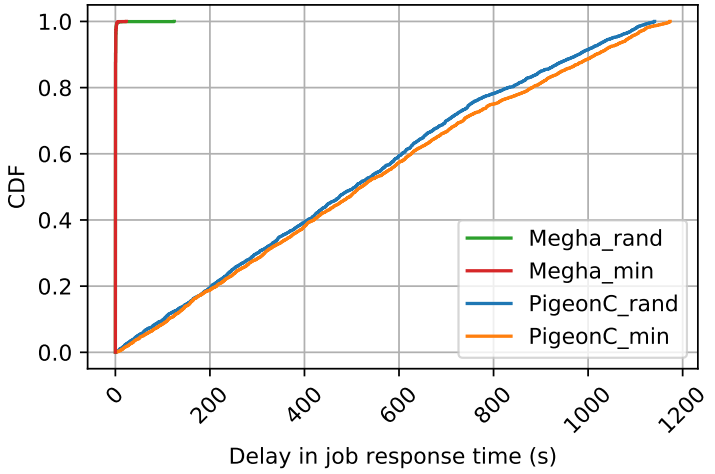
Under PigeonC, the median delays increased with an increase in the load exerted by the workload. For the Yahoo workload, the median delays were 1281.3 s under the Random approach and 1161.86 s under the Minimum Constraints approach. In contrast, the Google workload exhibited much lower median delays, around 1.5 ms under both approaches. The synthetic workloads displayed similar trends, with marginal differences in median delays. Specifically, for the syn_250 workload, median delays were 1.003 s (Random) and 1.002 s (Minimum Constraints), while for the syn_500 workload, the values were 507.74 s (Random) and 520.60 s (Minimum Constraints). The syn_1000 workload experienced median delays of 1416.88 s (Random) and 1449.3 s (Minimum Constraints). In the case of the Yahoo workload, the Minimum Constraints approach reduced the 99th percentile tail latency by approximately 1.05× (∼47,000 s). Meanwhile, for the Google workload and the syn_500 workload, the Random approach achieved a marginal improvement with reductions by factors of 1.002 (∼200 s) and 1.03 (∼30 s), respectively. Conversely, for the syn_1000 workload and the syn_250 workload, the Minimum Constraints approach slightly outperformed the Random approach with reductions by factors of 1.006 and 1.08, respectively. Notably, the Random approach led to an 8% reduction in the 99th percentile delays for the syn_250 workload, although the absolute value of the difference was only 0.3 s.

Under Megha, the median delay was consistently 1.5 ms in all scenarios, except for syn_1000, where it was 0.67 s (Random) and 0.33 s (Minimum Constraint). The Yahoo workload exhibited a 30% reduction (∼2000 s) in the 99th percentile delays under the Random approach compared to the Minimum Constraints approach for one seed value, while for other seeds, the difference was insignificant. For the syn_250, syn_500, and syn_1000 workloads, the 99th percentile delay differences between the two approaches were less than 1s, with a minimal 0.2% reduction for the Google workload under the Minimum Constraints approach.

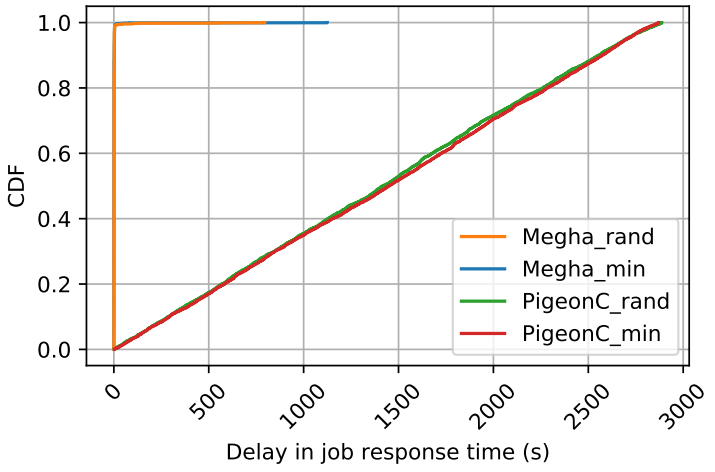
Compared to PigeonC, Megha improved the 99th percentile delays in short jobs by a factor of 5.61 and 5.3 for the Yahoo and Google workloads, respectively. All the jobs in the synthetic traces were considered to be short jobs since the task durations for all the jobs were set to 1 s, therefore the short job performance is identical to the overall performance of the workload.



(**a**) syn_250 workload



(**b**) syn_500 workload



(**c**) syn_1000 workload

Figure 6. Performance of the synthetic workloads.

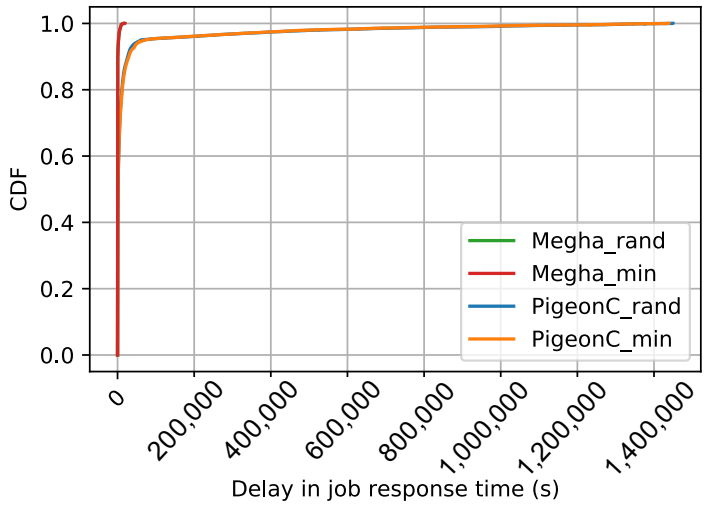


Figure 7. Performance of the Yahoo workload.

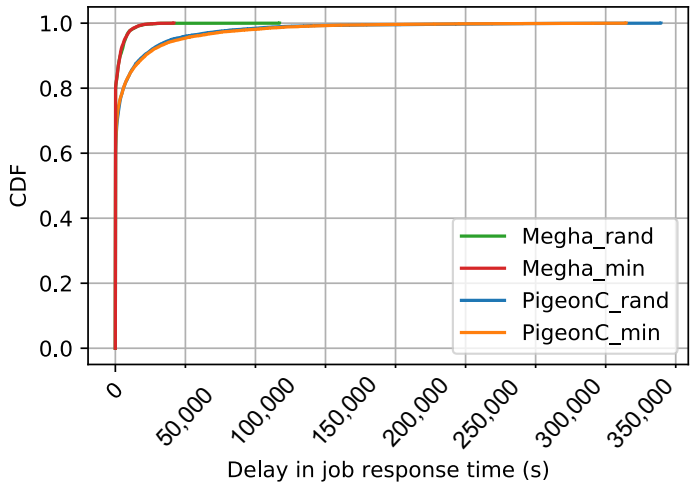


Figure 8. Performance of the Google workload.

The resource consumption of the workloads was recorded in every experiment and the resource utilization of the data center was calculated using Equation (2). For both Megha and PigeonC, the difference in the utilization for the two matching approaches was consistently less than 0.5%. We have, therefore, taken an average of the utilization reported for each scheduler across all runs, regardless of the constraint-matching approach employed. The mean utilization of PigeonC for each of the workloads, normalized with respect to Megha's mean utilization, has been represented in Figure 9. To show the distribution of the utilization, Figure 10 shows the CDF of the utilization under Megha and PigeonC. Since the utilization varies insignificantly over the simulation runs with different seed values and constraint-matching approaches, we have only shown the CDFs for one seed value and the Random approach.

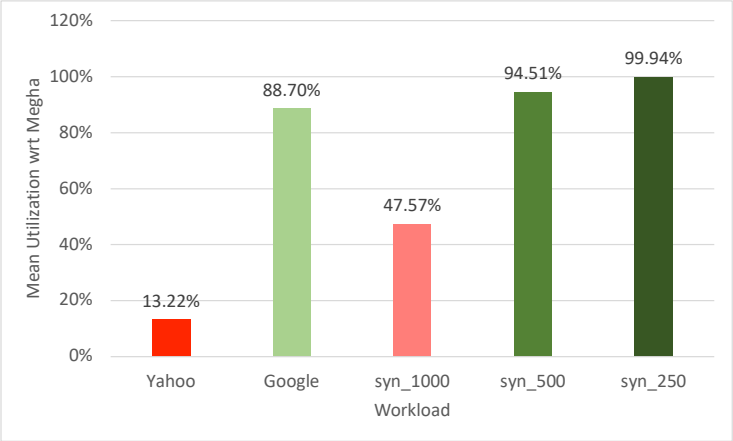
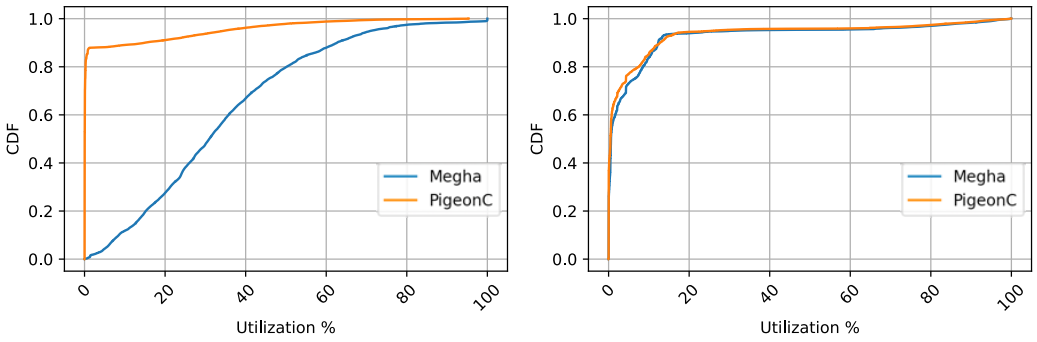
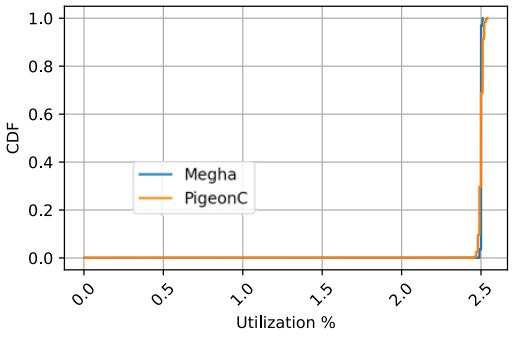


Figure 9. PigeonC's utilization normalized with respect to Megha's utilization

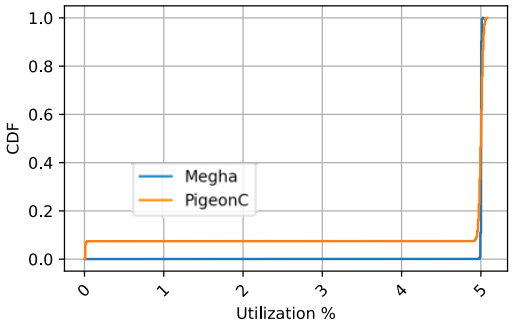


(**a**) Yahoo workload

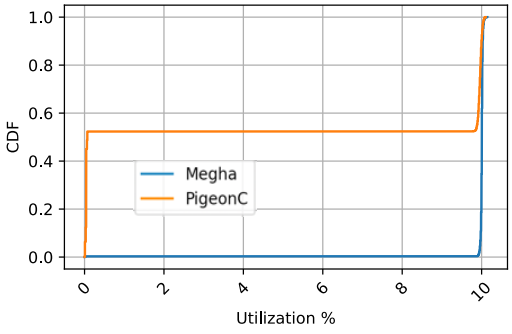
(**b**) Google subtrace workload



(**c**) syn_250 workload



(**d**) syn_500 workload



(**e**) syn_1000 workload

Figure 10. Utilization of the synthetic workloads recorded under Megha and PigeonC.

7. Discussion

In this section, we discuss the performance in terms of delay in job response time under the two scheduling architectures and examine the performance of the workloads under the two matching approaches, while also considering the relevance to IoT applications. The key findings have been summarized in Table 2.

Table 2. Summary of Key Findings and Recommendations

Findings & Recommendations	Description
Finding 1: Megha Outperforms PigeonC	Our experiments consistently show that Megha outperforms PigeonC. The median job response time delay in Megha was at least an order of magnitude lower than in PigeonC.
Finding 2: Significant Variation Seen in Job Response Time under Both Schedulers	The 99th percentile of the delay in job response time was found to be 3 to 6 orders of magnitude higher than the median delay.
Finding 3: Megha Reports Higher Utilization Compared to PigeonC	Across all workloads, Megha recorded a higher average utilization than PigeonC. The difference was more prominent for workloads exerting higher loads.
Finding 4: Megha Performs Better with Random Approach	For the Yahoo workload, which exerts the highest load, the delays reported under the Minimum Constraints approach were 1.29x higher than the Random approach. For other workloads, the difference was not significant.
Finding 5: PigeonC Performs Better with Minimum Constraints Approach	For the Yahoo workload, PigeonC recorded a 10% improvement in job delays under the Minimum Constraints approach over the Random approach. The difference was not significant for the other workloads.
Recommendation: Using Megha with the Random Approach for IoT workloads	Scheduling IoT workloads with Megha will result in consistent and predictable delays and higher utilization of resources.

Finding 1: Megha Outperforms PigeonC

The experimental results consistently demonstrate that Megha outperforms PigeonC across all workloads, irrespective of the matching approach used. Specifically, the median job response time delay is at least an order of magnitude lower in Megha when compared to PigeonC. PigeonC exhibited a wide range of job response time delays, spanning from milliseconds to the order of 10^6 s. In contrast, Megha maintained job response time delays below 1.5×10^4 s, with the highest delays observed in workloads with greater loads.

This performance gap can be attributed to several key factors. In Megha, GMs schedule tasks with global resource availability knowledge, and flexibility to place tasks anywhere in the DC. In contrast, PigeonC's Distributors and Masters do not have the same degree of flexibility and resource awareness. The Distributors are fast, load-balancing entities that work in parallel to distribute the tasks to different Masters. Supplying Distributors with up-to-date, complete resource availability information would impose a significant burden in terms of communication and processing overhead. Therefore, PigeonC's Distributors rely on the distribution of constraints within Masters' clusters when assigning tasks. Although Masters in PigeonC possess accurate and current resource availability information within their clusters, once a task reaches a Master, it is confined to running on resources available solely within that Master's cluster. Consequently, even when suitable worker nodes exist in other clusters capable of satisfying a task's requirements, the task is unable to utilize those resources and must wait until resources become available within its designated cluster. This limitation becomes more pronounced as the per-constraint load increases, leading to a rise in conflicts. A higher number of conflicts further amplifies the queuing delays experienced by tasks, contributing to overall job response time delays within PigeonC.

In summary, Megha's superior performance can be attributed to its GMs having access to global resource availability information and task placement flexibility. These two properties help Megha handle higher loads more effectively. PigeonC encounters difficulties in high-load scenarios primarily due to the limited scheduling scope for tasks within individual clusters, which leads to an increase in conflicts and queuing delays.

Finding 2: Significant Variation Seen in Job Response Time under Both Schedulers

Under both scheduling architectures, there was considerable variation in the delays experienced by the jobs in the workloads. While the median of the delay in job response time remained within a limit of 700 ms and ∼3000 s, under Megha and PigeonC, the 99th percentile delay was consistently 3 to 6 orders of magnitude higher than the median delay. This phenomenon can be attributed to multiple factors. Firstly, in the case of the real-world workloads, the difference in median and 99th percentile load is significant. When the load is less, the jobs may encounter insignificant delays; however, when the load increases, it causes delays to cascade, leading to larger queuing delays in jobs that are queued and in future jobs. Second, as explained earlier, the presence of constraints restricts the number of available resources for each task, thereby further increasing the ratio of demand to supply, which results in larger queuing delays. Third, a minimum delay in tasks due to communication overhead cannot be avoided. Although this delay may appear insignificant on its own, it can compound over time. This cumulative effect occurs as each task that waits on a resource adds its own portion of the minimum delay to the queue, impacting all subsequently scheduled tasks that require the same resource.

Finding 3: Megha Reports Higher Utilization Compared to PigeonC

As previously discussed, delays in job response time directly impact the resource utilization of the data center. In all the scenarios, Megha consistently reported a higher average utilization compared to PigeonC. The limited flexibility in PigeonC led to more idle resources, resulting in a lower utilization when compared to Megha. However, for the workloads with lower resource demands per second (as in the case of syn_250 and syn_500 workloads), the difference in utilization was less pronounced. This is because when task resource demands are lower, PigeonC can successfully accommodate each task within its assigned cluster without significant wait times, mitigating the impact of the inflexibility in task placement seen in PigeonC.

Finding 4: Megha Performs Better with Random Approach:

From the results of the experiments, it is apparent that the Yahoo workload, which exerts the highest load on the DC's resources, responds differently to the constraint-matching approaches compared to the other workloads. In the case of Megha, both constraint-matching approaches yielded identical median delays for the Yahoo workload. However, the 99th percentile delay was 1.29 times higher (approximately 2000 s) under the Minimum Constraints approach compared to the Random approach. This can be attributed to the higher load of the Yahoo workload (99th percentile load of approximately 1.1) and the reduced inconsistencies due to increased randomness, as discussed in Section 4.3.1.

Finding 5: PigeonC Performs Better with Minimum Constraints Approach

Under PigeonC, the results are reversed. For the Yahoo workload, the Minimum Constraints approach improved the 99th percentile delays by a factor of 1.1 (1.98×10^4 s). This is because, the scheduling entity in PigeonC, unlike Megha, possesses comprehensive and accurate information about resource availability in its clusters. Consequently, it does not need to address issues related to inconsistencies and conflicting scheduling decisions, which are mitigated by randomness. Instead, the Minimum Constraints approach in PigeonC reduces resource wastage of valuable machines with numerous constraints, enhancing the likelihood of future tasks finding suitable resources without queuing, thus reducing delays.

The difference in performance between the two approaches was less significant for the other workloads. For example, in the Google workload, which has a lower 99th percentile load than the Yahoo workload, the difference in the performance of Megha under the two approaches was not significant (ratio of 99th percentile delay approximately 0.98). This is because the advantages of the randomness approach were offset by resource wastage when selecting worker nodes with more constraints than required.

Recommendation:

Since IoT workloads are known to be unpredictable, we recommend using Megha, configured with the Random approach, to schedule the workloads in both edge and cloud data centers. This will make job response times more consistent and lower than with PigeonC. Additionally, it will also improve the utilization of the data center's resources. However, if the data center has little variation in workload patterns and rarely experiences high load, the difference in performance between Megha and PigeonC, with either constraint-matching approach, would not be significant.

8. Conclusions

To meet the demands of increasing data and processing complexity brought on by the proliferation of IoT and artificial intelligence applications, modern data centers must rely on heterogeneous infrastructure. To fully leverage the potential speed-ups offered by hardware such as GPUs and FPGAs, data center schedulers must be aware of infrastructure heterogeneity and the constraints imposed by the workloads.

This paper comprehensively explores federated scheduling architectures for heterogeneous workloads with task placement constraints. Two such architectures, PigeonC and Megha, are discussed in detail. PigeonC is a modified variant of federated scheduling architecture, Pigeon, with constraint-awareness, while Megha is enhanced for efficient handling of constraint-rich workloads. Additionally, we study the performance of two constraint-matching approaches, the Random and Minimum Constraints approaches.

Our experiments with real-world and synthetic cluster traces show that Megha reduces the 99th percentile delays in job response times of the workloads by at least a factor of 10 when compared to PigeonC. Megha, particularly when using the Random approach, outperforms the Minimum Constraints approach, yielding 1.29 times lower delays in job response times. Conversely, under high load conditions, PigeonC shows a 10% reduction in job response time delays when employing the Minimum Constraints approach compared to the Random approach. However, under lower loads, both approaches deliver similar levels of performance. We recommend that Megha be used with the Random constraint-matching approach for IoT workloads with stringent latency requirements.

In our future work, we would like to study the performance of federated architectures when scheduling workloads with constraints involving inter-task dependencies and data locality preferences.

Author Contributions: Conceptualization, S.K. and D.S.; methodology, D.S. and M.T.; software, M.T.; validation, M.T., S.K. and D.S.; formal analysis, M.T.; investigation, M.T.; data curation, M.T.; writing—original draft preparation, M.T.; writing—review and editing, S.K.; visualization, M.T.; supervision, D.S. and S.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Simulator code and cluster traces are available in a publicly accessible repository that does not issue DOIs: https://github.com/meghanat/megha-constraint-aware.git (accessed on 1 November 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dang, L.M.; Piran, M.J.; Han, D.; Min, K.; Moon, H. A survey on internet of things and cloud computing for healthcare. *Electronics* **2019**, *8*, 768. [CrossRef]
2. Zantalis, F.; Koulouras, G.; Karabetsos, S.; Kandris, D. A review of machine learning and IoT in smart transportation. *Future Internet* **2019**, *11*, 94. [CrossRef]
3. Farooq, M.S.; Riaz, S.; Abid, A.; Umer, T.; Zikria, Y.B. Role of IoT technology in agriculture: A systematic literature review. *Electronics* **2020**, *9*, 319. [CrossRef]
4. Xu, J.; Gu, B.; Tian, G. Review of agricultural IoT technology. *Artif. Intell. Agric.* **2022**, *6*, 10–22. [CrossRef]
5. Ning, H.; Li, Y.; Shi, F.; Yang, L.T. Heterogeneous edge computing open platforms and tools for internet of things. *Future Gener. Comput. Syst.* **2020**, *106*, 67–76. [CrossRef]

6. Tang, S. Performance Modeling and Optimization for a Fog-Based IoT Platform. *IoT* **2023**, *4*, 183–201. [CrossRef]
7. Tadakamalla, U.; Menascé, D.A. Autonomic resource management using analytic models for fog/cloud computing. In Proceedings of the 2019 IEEE International Conference on Fog Computing (ICFC), Prague, Czech Republic, 24–26 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 69–79.
8. Yamato, Y.; Demizu, T.; Noguchi, H.; Kataoka, M. Automatic GPU offloading technology for open IoT environment. *IEEE Internet Things J.* **2018**, *6*, 2669–2678. [CrossRef]
9. Kim, C.; Kim, S. Optimizing Logging and Monitoring in Heterogeneous Cloud Environments for IoT and Edge Applications. *IEEE Internet Things J.* **2023**, *early access*. [CrossRef]
10. Bian, J.; Al Arafat, A.; Xiong, H.; Li, J.; Li, L.; Chen, H.; Wang, J.; Dou, D.; Guo, Z. Machine learning in real-time internet of things (iot) systems: A survey. *IEEE Internet Things J.* **2022**, *9*, 8364–8386. [CrossRef]
11. Sharma, B.; Chudnovsky, V.; Hellerstein, J.L.; Rifaat, R.; Das, C.R. Modeling and synthesizing task placement constraints in Google compute clusters. In Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, 26–28 October 2011; pp. 1–14.
12. Thinakaran, P.; Gunasekaran, J.R.; Sharma, B.; Kandemir, M.T.; Das, C.R. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 977–987.
13. Wang, Z.; Li, H.; Li, Z.; Sun, X.; Rao, J.; Che, H.; Jiang, H. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, 20–23 November 2019; pp. 246–258.
14. Curino, C.; Krishnan, S.; Karanasos, K.; Rao, S.; Fumarola, G.M.; Huang, B.; Chaliparambil, K.; Suresh, A.; Chen, Y.; Heddaya, S.; et al. Hydra: A federated resource manager for data-center scale analytics. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), Boston, MA, USA, 26–28 February 2019; pp. 177–192.
15. Thiyyakat, M.; Kalambur, S.; Chaudhary, R.; Nayak, S.G.; Shetty, A.; Sitaram, D. Eventually-Consistent Federated Scheduling for Data Center Workloads. *arXiv* **2023**, arXiv:2308.10178.
16. Thiyyakat, M.; Kalambur, S.; Sitaram, D. Megha: Decentralized Federated Scheduling for Data Center Workloads. In Proceedings of the 2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS), Bangalore, India, 3–8 January 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 278–286.
17. Ousterhout, K.; Wendell, P.; Zaharia, M.; Stoica, I. Sparrow: Distributed, low latency scheduling. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farmington, PA, USA, 3–6 November 2013; pp. 69–84.
18. Delgado, P.; Didona, D.; Dinu, F.; Zwaenepoel, W. Job-aware scheduling in eagle: Divide and stick to your probes. In Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, 5–7 October 2016; pp. 497–509.
19. Chen, Y.; Ganapathi, A.; Griffith, R.; Katz, R. The case for evaluating mapreduce performance using workload suites. In Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, Singapore, 25–27 July 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 390–399.
20. Reiss, C.; Wilkes, J.; Hellerstein, J.L. *Google Cluster-Usage Traces: Format+ Schema*; White Paper; Google Inc.: Menlo Park, CA, USA, 2011; Volume 1, pp. 1–14.
21. Delgado, P.; Dinu, F.; Kermarrec, A.M.; Zwaenepoel, W. Hawk: Hybrid datacenter scheduling. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15), Santa Clara, CA, USA, 8–10 July 2015; pp. 499–510.
22. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.; Shenker, S.; Stoica, I. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), Boston, MA, USA, 25–27 April 2011.
23. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
24. Hao, C.; Shen, J.; Chen, C.; Zhang, H.; Wu, Y.; Li, M. Pcssampler: Sample-based, private-state cluster scheduling. In Proceedings of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 14–17 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 599–608.
25. Reiss, C.; Tumanov, A.; Ganger, G.R.; Katz, R.H.; Kozuch, M.A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the Third ACM Symposium on Cloud Computing, San Jose, CA, USA, 14–17 October 2012; pp. 1–13.
26. Delimitrou, C.; Kozyrakis, C. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Not.* **2013**, *48*, 77–88. [CrossRef]
27. Cuomo, A.; Di Modica, G.; Distefano, S.; Puliafito, A.; Rak, M.; Tomarchio, O.; Venticinque, S.; Villano, U. An SLA-based broker for cloud infrastructures. *J. Grid Comput.* **2013**, *11*, 1–25. [CrossRef]
28. Garefalakis, P.; Karanasos, K.; Pietzuch, P.; Suresh, A.; Rao, S. Medea: Scheduling of long running applications in shared production clusters. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; pp. 1–13.
29. Li, S.; Wang, L.; Wang, W.; Yu, Y.; Li, B. George: Learning to place long-lived containers in large clusters with operation constraints. In Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, 1–3 November 2021; pp. 258–272.
30. Tumanov, A.; Zhu, T.; Park, J.W.; Kozuch, M.A.; Harchol-Balter, M.; Ganger, G.R. TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In Proceedings of the Eleventh European Conference on Computer Systems, London, UK, 18–21 April 2016; pp. 1–16.

31. Swain, C.K.; Gupta, B.; Sahu, A. Constraint aware profit maximization scheduling of tasks in heterogeneous datacenters. *Computing* **2020**, *102*, 2229–2255. [CrossRef]
32. EPFL Labos. Hawk/Eagle Simulator. 2017. Available online: https://github.com/epfl-labos/eagle/tree/master/simulation (accessed on 11 September 2023).