Falcon: An Efficient Dependency-Aware Scheduling for High Throughput and Resource Utilization in Clouds

Jinwei Liu*, Rui Gong†, Richard A. Alo*, Wei Dai‡, Richard A. Long§, Pierre Ngnepieba§
*Department of Computer and Information Sciences, Florida A&M University, Tallahassee, FL 32307, USA
†Department of Informatics and Mathematics, Mercer University, Macon, GA 31207, USA
‡Department of Computer Science, Purdue University Northwest, Hammond, IN 46323, USA
§College of Science and Technology, Florida A&M University, Tallahassee, FL 32307, USA
*{jinwei.liu, richard.alo}@famu.edu, †gong_r@mercer.edu, ‡weidai@pnw.edu,
§{richard.long, pierre.ngnepieba}@famu.edu

***Abstract*—Diverse workloads in modern datacenters increasingly comprise data parallel jobs. Production data parallel jobs have complex dependency structure (e.g., complex task dependencies) and heterogeneous resource demands. It is challenging to design a scheduler with high throughput for the data parallel applications in clouds. Preemption is also necessary to provide short waiting time for high priority jobs/tasks. Previous studies on scheduling and preemption do not fully utilize task dependency to increase throughput, and they usually have high overhead caused by the preemption (time overhead of context switching). To address this challenge, we propose Falcon, an efficient dependency-aware scheduling for high throughput and resource utilization as well as low overhead in clouds. Falcon utilizes task dependency information to determine tasks' priorities for reducing the overhead caused by preemption, and it also packs complementary tasks (tasks whose demands on multiple resource types are complementary to each other) for improving the resource utilization. Extensive testbed results based on a real cluster and experiments using real-world Amazon EC2 cloud service show that Falcon has better performance on throughput and overhead compared to existing strategies.**

***Index Terms*—scheduling, task dependency, throughput, resource utilization, priority, preemption**

I. Introduction

The resources and workloads in production environment can be both heterogeneous and dynamic [1–3]. Production datacenters usually run vast number of applications with diverse characteristics [4], and they encounter increasingly heterogeneous workloads [2, 5–7]. In Microsoft production clusters, the durations of tasks can vary from a few milliseconds to tens of thousands of seconds [7]. The heterogeneity and dynamicity of workloads not only challenges the scheduling efficiency (e.g., performance improvement on latency and throughput), but also poses a challenge for improving resource utilization [2, 8, 9]. Public safety requires emergency response that is timely and efficient. Efficient scheduling of tasks (e.g., live video stream tasks) can help enhance public safety by powering faster responses, enhancing situational awareness, optimizing resource utilization, strengthening disaster recovery, and upholding data security and compliance [10, 11].

In parallel applications, the dependency between concurrent tasks (e.g., tasks with execution time 100ms) is increasingly common. Previous scheduling approaches [12–15] cannot well handle sub-second parallel jobs with task dependency (a relationship that requires a particular order for tasks to be executed) constraints. They cannot fully utilize task dependency to increase throughput and satisfy high-priority jobs/tasks while reducing overhead.

Different jobs/tasks have different priorities, and high priority jobs/tasks should be served first. To satisfy high priority jobs/tasks, priority based scheduling is necessary, and preemption is also needed when a high priority task arrives at a machine running a lower priority task. However, preemption amplifies the scheduling challenge [15] (i.e., Preemption incurs time overhead of context switching). Satisfying high priority jobs/tasks with less time overhead still has not been well addressed. Previous works [14, 16–18] adopt priority-based scheduling to handle the problem of satisfying high priority jobs, however previous works either cannot satisfy high priority jobs/tasks well or incur time overhead caused by preemption. The work [16] proposes Jockey, a priority-based scheduling, and provides latency SLOs for data parallel jobs written in SCOPE with Jockey. The work [14] proposes a priority-based low latency distributed scheduling. However, both [16] and [14] do not support preemption, and thus they cannot truly satisfy high priority tasks because the high priority tasks have to wait until the low priority tasks complete. Although the work [18] presents Natjam, a system that supports arbitrary job priorities and preemption for MapReduce clusters, Natjam can easily incur overhead caused by preemption because it simply preempts a job or task based on only the priorities of the waiting job (task) and the running job (task). Amoeba [17] provides instantaneous fairness with elastic queues, and uses a checkpointing mechanism for preemption. Although tasks execution can be resumed at safe points, Amoeba can still incur overhead because it checkpoints the longest-running tasks without the consideration of other waiting tasks.

To address this problem, in this paper, we aim to develop Falcon, an efficient dependency-aware scheduling, which can increase the throughput and resource utilization in clouds. Falcon outperforms previous schedulers in that it can well handle the scheduling of jobs with dependency constraints (task dependency), and it can satisfy high-priority tasks and reduce the time overhead caused by preemption. We summarize the contributions of this work as follows.

- We propose Falcon, an efficient dependency-aware scheduling, which can increase the throughput and resource utilization

while reducing the overhead in clouds.

- Falcon splits jobs into tasks by taking into account task dependency, and assigns the tasks that do not depend on each other to different workers (or different cores of a worker) so that these tasks can be processed in parallel and the response time can thus be reduced.
- Falcon leverages the complementary of tasks' requirements on different resource types, and it packs complementary tasks and allocates them to a worker for improving the resource utilization.
- We propose Selective Preemption (SP), which can satisfy high-priority tasks and reduce the time overhead caused by preemption.

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

II. Related Work

A. Throughput Optimization

Many methods have been proposed for improving throughput in scheduling. Qiao *et al.* [19] proposed Pollux to improve the cluster-wide throughput. Gu *et al.* [20] proposed Tiresias, which adopts the Least-Attained Service and Gittins Index algorithm to increase the job throughput. Amaro *et al.* [21] proposed faster swapping mechanisms and a far memory-aware cluster scheduler. They examined the conditions under which this use of far memory can increase job throughput. Vijayakumar *et al.* [22] proposed a decentralized scheduler, Murmuration, that can reduce the total wait time of tasks and increase the throughput of jobs in the system. Unlike the above methods, Falcon considers the priorities of jobs (tasks) and uses Selective Preemption to satisfy high-priority tasks and reduce the time overhead caused by preemption.

B. Improving Resource Utilization

Wang *et al.* [2] proposed a two-layer, hierarchical scheduler for achieving low latency for short jobs while maintaining high resource utilization. Xiang *et al.* [23] proposed a resource management and scheduling system GÖDEL, which co-locates various workloads on each machine to achieve better resource utilization and elasticity. Chang *et al.* [24] presented Eva, a cloud-based cluster scheduler designed to serve batch processing workloads cost-efficiently. Eva uses a reservation price-based scheduling algorithm to jointly optimize task assignment and instance provisioning for improving resource utilization and reducing cost. Guo *et al.* [25] proposed vSched, which probes accurate vCPU abstraction through a set of lightweight microbenchmarks (vProbers), and leverages the probed information to optimize task scheduling in cloud VMs. vSched leverages intra-VM harvesting (IVH) to improve the resource utilization (vCPU utilization).

In our proposed method, Falcon splits jobs into tasks by taking the constraints of tasks into account, and assigns the tasks that do not depend on each other to different machines (or different cores of a machine) so that these tasks can run in parallel and the throughput can be increased. Falcon also takes the constraints of jobs (tasks) on resources into account and schedules jobs (tasks) with the consideration of improving resource utilization by packing complementary tasks. Finally, Falcon considers the priorities of different jobs (tasks) and uses Selection Preemption to satisfy hight priority tasks while reducing overhead.

III. System Model

In this section, we first introduce some concepts and assumptions, then we formulate a research problem. Finally, we describe our proposed algorithms for improving resource utilization and reducing time overhead caused by preemption.

A. Concepts and Assumptions

A job is supposed to be split into n_t tasks, and the tasks are assigned to workers (e.g., virtual machines) based on tasks' resource demands and task dependency. We assume jobs can be handled by any scheduler and tasks are run by workers in a fixed number of slots. A buffer queue is used to queue tasks assigned to a worker when the worker cannot run those tasks concurrently. We define *makespan* (or schedule length) as the time when all jobs finish processing and *throughput* as the total number of jobs that complete their execution per time unit.

Problem Statement: Given a set of jobs consisting of tasks, resource demands of each job/task, constraints of tasks in each job (e.g., task precedence constraints), a number of heterogeneous worker machines, and their resource capacity constraints, what is the makespan? Then, how to design an efficient scheduler to schedule these jobs so that the throughput can be maximized while satisfying high priority tasks and reducing time overhead?

B. Scheduling

Resource scheduling in cloud computing is an NP-hard problem and has a high computational complexity [24, 26–28]. Thus, we propose a heuristic method called Falcon. Falcon first accurately estimates the run time of jobs [29], and it then utilizes the method in the work [9] to classify jobs into long and short jobs based on the estimated run time of jobs based on the extracted features.

1) Improving Resource Utilization: Resource fragmentation can cause poor resource utilization, and it is crucial to reduce resource fragmentation [15, 24, 30, 31]. In Microsoft Azure, even 1% in fragmentation reduction can lead to cost savings in the order of $100M per year [32]. To improve resource utilization in clouds, Falcon presents a task packing strategy.

Falcon first packs the tasks with complementary dominant resources (A dominant resource is defined as the resource type on which the task has the highest demand) such that the summation of the deviation of the two tasks' resource demands on each resource type is the largest (refer to Eq. (1)). Given a list of tasks, Falcon fetches each task T_i, and tries to find its complementary task from the list to pack with T_i. To find T_i's complementary task, Falcon uses Eq. (1) to calculate its deviation with every other task T_j with a different dominant resource.

$$DV(j,i) = \sum_{h=1}^{l}((d_{jh} - \frac{d_{jh}+d_{ih}}{2})^2 + (d_{ih} - \frac{d_{jh}+d_{ih}}{2})^2), \quad (1)$$

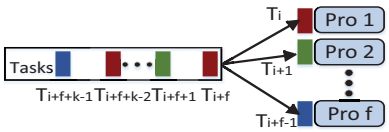


Fig. 1: Preemption for multiple tasks running on multiple processors.

where d_{ih} is T_i's resource demand on resource type h (e.g., CPU). Finally, the task with the highest deviation value is the complementary task of T_i. This method can also be applied to task packing for more tasks (refer to the work [9] for more details).

2) Resource Allocation: After task packing, Falcon assigns each task entity (packed tasks or a task) to a server that meets the resource demand of the task entity and has the least remaining resources (called most matched server).

C. Selective Preemption

To reduce the waiting time for high priority tasks and time overhead caused by preemption, we propose *Selective Preemption* (SP). It chooses a task among the waiting tasks with the top priority range with the probability equaling the task's normalized priority which is the ratio of the reverse rank of a task's priority among all the tasks in the queue to the number of tasks in the queue.

1) Task Priority Determination: Given a particular and arbitrary task T_{ij}, the priority of task T_{ij} at time t can be recursively calculated as follows:

$$P_{ij}^t = \sum_{T_{ik} \in S_{ij}} P_{ik}^t, \tag{2}$$

where S_{ij} represents a set consisting of T_{ij}'s children. For a given task (e.g., T_{ij}) that has no dependent tasks, its priority at time t can be calculated as follows:

$$P_{ij}^t = \omega_1 \cdot 1/t_{ij}^{rem} + \omega_2 \cdot t_{ij}^w, \tag{3}$$

where t_{ij}^{rem} and T_{ij}^w are task T_{ij}'s remaining time and waiting time, respectively. ω_1 and ω_2 are the weights for the task's remaining time and waiting time, respectively, and $\omega_1 + \omega_2 = 1$.

Given a worker machine m_i, without loss of generality, suppose there are f running tasks $(T_i, ..., T_{i+f-1})$, and there are k tasks waiting $(T_{i+f}, ..., T_{i+f+k-1})$ in the queue of m_i (see Figure 1). Given a task, the task (say T_{i+f}) can execute only if the following three conditions are satisfied:

- Condition C_1: The priority of the waiting task T_{i+f} is higher than that of the running task, i.e., $P_{i+f} > P_j$ $(j \in \{i, i+1, ..., i+f-1\})$.
- Condition C_2: The waiting task T_{i+f} is independent of the running task T_j $(j \in \{i, i+1, ..., i+f-1\})$.

To understand Condition C_3, we first introduce a theorem.

Theorem 4.1. *Given a server with f processors, if f tasks $(T_i, ..., T_{i+f-1})$ are running, and k tasks $(T_{i+f}, ..., T_{i+f+k-1})$ are waiting in the queue of the server, then at most the top $\lfloor (1-\delta)k \rfloor$ priority tasks are allowed for preemption for each processor, where δ is the minimum required ratio between 0 and 1.*

Proof: According to Formula (5), at least $\lceil \delta k \rceil$ tasks' priorities are lower than P_{i+f} if the preemption is allowed for the task T_{i+f}, that is, it allows preemption for at most the top $k - \lceil \delta k \rceil$ tasks for each processor. Thus at most the top $\lfloor (1-\delta)k \rfloor$ priority tasks are allowed for preemption. ■

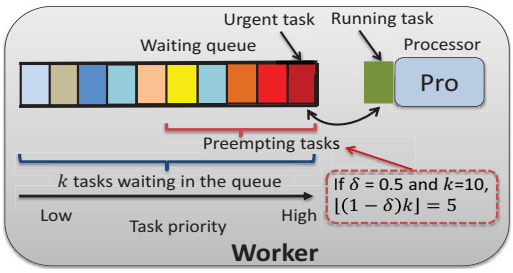


Fig. 2: An example illustrating the process of selecting a waiting task to preempt the running task in Falcon.

- Condition C_3: The rank of the waiting task's priority among all of the waiting tasks in the queue is top $\lfloor (1-\delta)k \rfloor$ (based on Theorem 4.1) for each processor.

We calculate the priorities of the running tasks $T_i, ..., T_{i+f-1}$ and the waiting tasks $T_{i+f}, ..., T_{i+f+k-1}$ using Eqs. (2) and (3). Define I as an indicator function. For any two tasks T_i and T_j,

$$I_{ij} = \begin{cases} 1, & P_i > P_j \\ 0, & P_i \leq P_j, \end{cases} \tag{4}$$

where P_i and P_j are the priorities of task T_i and task T_j.

For a particular and arbitrary running task T_i in worker m_i, we check if Condition C_3 is satisfied using the following formula:

$$\frac{\sum_{j=i+f}^{i+f+k-1} I_{i+f,j}}{k} \geq \delta, \tag{5}$$

where δ (minimum required ratio between 0 and 1) is related to the practical application, largely depending on the goal of the system. In the experiments, we set it to 0.35. To understand how Formula (5) works, we give an example to illustrate the process of selecting a waiting task to preempt the running task (see Figure 2). Suppose there are 10 tasks $(T_1, ..., T_{10})$ in the waiting queue of a worker, and their priorities are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. According to Formula (5), the priority of the waiting task should be no less than $\lceil 10\delta \rceil$ other waiting tasks. Also the waiting task should be independent of the running tasks, thus at most $\lfloor 10(1-\delta) \rfloor$ tasks are allowed for preemption. If $\delta = 0.5$, then at most top 5 priority tasks are allowed for preemption.

IV. System Design

In this section, we introduce the design of Falcon.

A. Architecture of the Scheduler

Figure 3 shows the architecture of Falcon. When a user submits a job to the cloud system, the system will deliver the job to the scheduler that is not heavily loaded and has the smallest geographic distance to the user. The scheduler first splits the job into tasks, then it distributes the tasks to r randomly selected masters that are not heavily loaded with the consideration of task dependency. The masters pack tasks that have complementary resource requirements based on the task packing strategy in Section III-B1, and assign task entity to the workers. The workers choose tasks for execution based on the Selective Preemption method (Algorithm 1 shows the scheduling of tasks). The workers periodically report their resource information to their master, and the master periodically updates a table which records the resource information of each worker associated with the master.

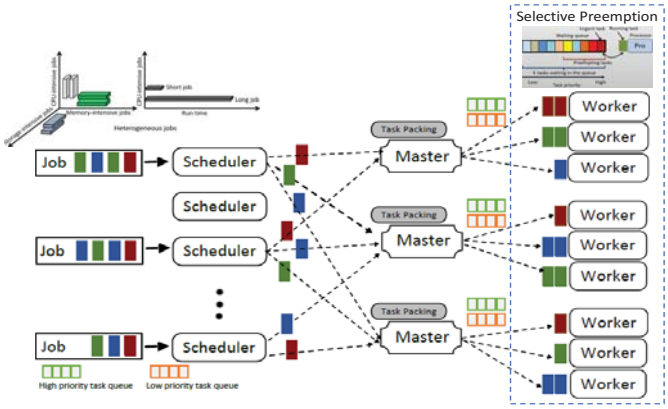


Fig. 3: Architecture of Falcon.

Algorithm 1: Task_Scheduling()

1 Each master packs complementary tasks based on the task packing strategy // Section III-B
2 **for** $i \leftarrow 1$ *to* m **do** // m denotes # of masters
3 Assign the task entity to a worker using the resource allocation algorithm // Section III-B
4 Execute tasks based on the Selective Preemption method

V. Performance Evaluation

We first conducted testbed experiments in a real cluster. We tested various evaluation metrics and compared our method with four methods. To further evaluate the performance of our method, we conducted experiments on the real-world Amazon EC2 [33]. In the following, we introduce our experimental results on a real cluster and the experimental results on Amazon EC2, respectively.

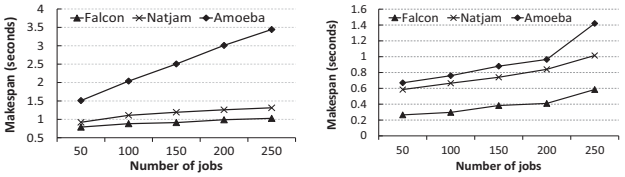
A. Experimental Results on Real Cluster

We deployed our testbed in a large-scale cluster [34], and implemented our method and other four methods in our testbed. We compared the results of our method and the other four methods Amoeba [17], Natjam [18], SRPT [35] and SNB [36] in various scenarios. We used up to 1,000 heterogeneous jobs which have different resource requirements, and we requested 4 schedulers, 6 masters and 50 workers. We used the Google cluster trace data [37] to set the parameters. The Google cluster trace [37] records resource usage on a cluster of about 11,000 machines from May 2011 for 29 days. We randomly chose tasks from the jobs in the period between May 1 to May 7. Table I shows the parameter settings in our experiments unless otherwise specified. First, we submitted jobs at a fixed rate and collected the results, then we varied the job submission rate and conducted the experiments again to test the effects of job submission rate on schedulers.

Figure 4(a) shows the relationship between the makespan and the number of jobs. In Figure 4(a), we see that the makespan increases as the number of jobs increases. This is because with a fixed number of workers, the more the jobs submitted, the more time the workers require to finish executing all the jobs. Moreover, we also observe that Falcon has the smallest makespan, and the makespans of Natjam, Amoeba and Falcon follow Falcon<Natjam<Amoeba. The

TABLE I: Parameter settings.

Parameter	Meaning	Setting
n	# of servers	10-50
h	# of jobs	50-1000
n_t	# of tasks of a job	10-20
δ	Minimum required ratio	0.35
ω_1	Weight for task's remaining time	0.5
ω_2	Weight for task's waiting time	0.5

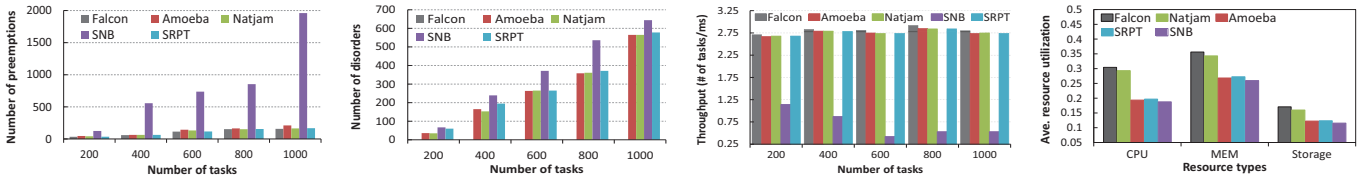


(a) Makespan for various methods with 10 workers (b) Makespan for various methods with 50 workers

Fig. 4: Performance of various methods on makespan.

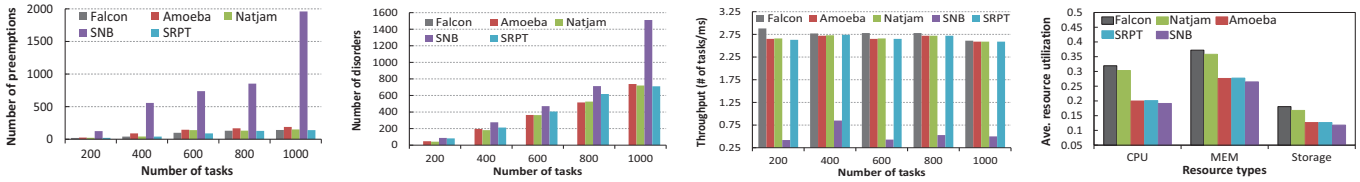
reason behind this is that Falcon takes into account task dependency and assigns tasks that are independent of each other to different workers (or different cores of a worker) so that the tasks can run in parallel, and thus reduces the response time of jobs and decreases the makespan. Also, Falcon uses the SP method to reduce the time overhead caused by preemption. However, both Natjam and Amoeba assign tasks to workers without the consideration of task dependency, which can increase time overhead. Although Natjam has deadline constraints for jobs, it neglects the task dependency, and it thus leads to time overhead caused by communication between different tasks. Figure 4(b) shows the relationship between the makespan and the number of jobs with 50 workers. In Figure 4(b), we observe the similar results due to the same reasons.

Figure 5 compares the performance of various evaluation metrics of Falcon with Amoeba, Natjam, SNB and SRPT with job submission rate 5 jobs per second. Figure 5(a) shows the relationship between overhead and the number of tasks. In Figure 5(a), we use the number of preemptions to represent the overhead because the overhead is mainly caused by preemption (e.g., context switch). We see that SNB has the highest overhead. As the number of tasks increases, the overhead of SNB increases. The overhead of Falcon in general is the lowest. This is because Falcon considers the rank of the waiting task's priority among all the waiting tasks in the queue and it allows preemptions for at most top $\lfloor (1-\delta)k \rfloor$ priority tasks. Figure 5(b) shows the relationship between the number of disorders for executing tasks that should have been executed sequentially due to the dependencies among them and the number of tasks. In Figure 5(b), we find the number of disorders follows Falcon<Natjam<Amoeba<SRPT<SNB. The reason behind this is that Falcon considers the dependencies among tasks when it schedules tasks, however, the other methods neglect the dependencies among tasks when they schedule tasks. Figure 5(c) shows the relationship between throughput (# of tasks/ms) and the number of tasks. In Figure 5(c), we see the throughput follows SNB<SRPT<Amoeba≈Natjam<Falcon. This is because (1) Falcon considers task dependency and schedules tasks that do not depend on each other to different processors or different machines to run tasks in parallel so that it decreases



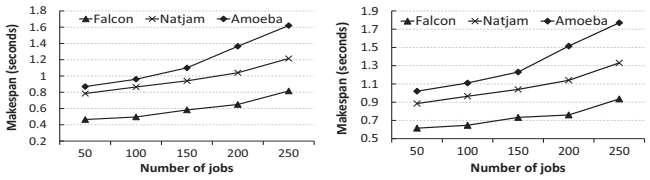
(a) Relationship between overhead and the number of tasks (b) Relationship between the number of disorders and the number of tasks (c) Relationship between throughput and the number of tasks (d) Relationship between resource utilization and the number of tasks

Fig. 5: Performance of various evaluation metrics of different methods with job submission rate 5 jobs/sec and 30 workers on a real cluster.



(a) Relationship between overhead and the number of tasks (b) Relationship between the number of disorders and the number of tasks (c) Relationship between throughput and the number of tasks (d) Relationship between resource utilization and the number of tasks

Fig. 6: Performance of various evaluation metrics of different methods with job submission rate 50 jobs/sec and 30 workers on a real cluster.



(a) Makespan for various methods with 20 workers (b) Makespan for various methods with 30 workers

Fig. 7: Makespan for various methods with 30 workers with job submission rate 5 jobs/sec on Amazon EC2.

response time and thus increases throughput; (2) Falcon uses the SP method for preemption, and it allows preemptions for at most top $\lfloor(1-\delta)k\rfloor$ priority tasks, which reduces the time overhead caused by preemption and thereby increases the throughput. Figure 5(d) shows the relationship between the average resource utilization and the number of tasks. In Figure 5(d), we see that the average resource utilization in general follows SNB<Amoeba≈SRPT<Natjam<Falcon. This is because Falcon packs complementary tasks and allocates them to workers, which can improve the resource utilization.

Figure 6 compares the performance of various evaluation metrics of Falcon with Amoeba, Natjam, SNB and SRPT with job submission rate 50 jobs per second. In Figure 6, we observe the similar results due to the same reasons. Both Figure 5 and Figure 6 suggest Falcon in general outperforms the other four methods.

B. Real-world Experimental Results

To fully test the performance of our method, we also conducted experiments on Amazon EC2. We used the similar heterogeneous jobs, 3 schedulers, 5 masters and 6 workers for each master. Each worker consists of 12GB memory and six 2.5 GHz processors. We deployed the implementation of our method and the other methods to the workers. We collected all results and calculated the average values of each metric.

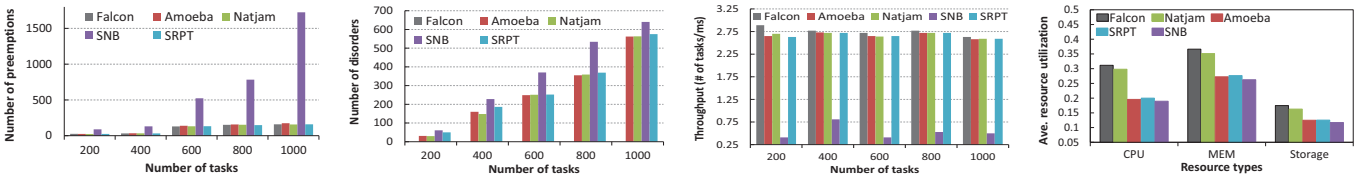
Figure 7(a) and Figure 7(b) show the relationship between the makespan and the number of jobs with 20 workers and 30 workers on Amazon EC2, respectively. In Figure 7(a) and Figure 7(b), we observe the similar results due to the same reasons. By examining Figure 7 and Figure 4, we see that the makespan in Figure 7(a) is relatively lower than that in Figure 4(a), and the makespan in Figure 4(b) is relatively lower than that in Figure 7(b). This is because there are more workers in Figure 7(a) and Figure 4(b).

Figure 8 compares the performance of various evaluation metrics of Falcon with Amoeba, Natjam, SNB and SRPT with job submission rate 5 jobs per second on Amazon EC2. The results in general are consistent with our testbed results. By examining Figure 5, Figure 6 and Figure 8, we find both our testbed results and real-world experimental results on EC2 show our method in general performs better than the other four methods.

Figure 9 compares the performance of various evaluation metrics of Falcon with Amoeba, Natjam, SNB and SRPT with job submission rate 50 jobs per second on Amazon EC2. The results in Figure 9 in general are consistent with our testbed results.

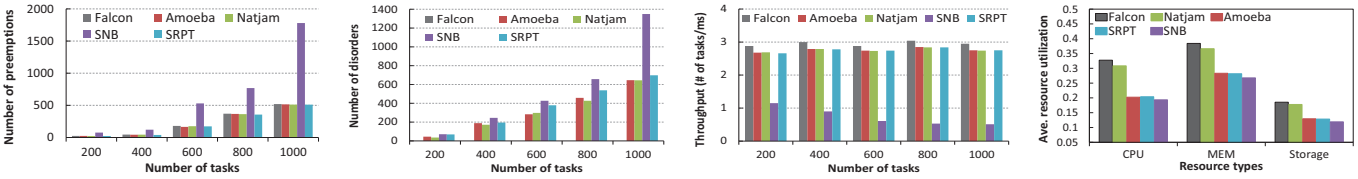
VI. Conclusions

This paper presents Falcon, an efficient dependency-aware scheduling for high throughput and resource utilization as well as low overhead in clouds. Falcon considers task dependency and reduces the response time of jobs by running tasks that are independent of each other in parallel. To satisfy the high priority tasks without incurring much overhead, Falcon presents Selective Preemption which can reduce the waiting time of high priority task and the time overhead caused by preemption. Moreover, Falcon assigns tasks to workers with the consideration of improving resource utilization. We compare our method with the existing methods under various scenarios using a large real cluster and Amazon EC2 cloud service, and demonstrate Falcon outperforms the exiting methods under both the real cluster and Amazon EC2 cloud service. In the future, we will use different cloud/cluster workloads (including machine learning workloads or GPU-based workloads) to fully verify the performance of Falcon. Also, we will consider data locality, fairness, tail latency, and resource harvesting. In addition, we will consider fault tolerance and energy efficiency in designing a robust and efficient scheduling system.



(a) Relationship between overhead and the number of tasks (b) Relationship between the number of disorders and the number of tasks (c) Relationship between throughput and the number of tasks (d) Relationship between resource utilization and the number of tasks

Fig. 8: Performance of various evaluation metrics of different methods with job submission rate 5 jobs/sec and 30 workers on Amazon EC2.



(a) Relationship between overhead and the number of tasks (b) Relationship between the number of disorders and the number of tasks (c) Relationship between throughput and the number of tasks (d) Relationship between resource utilization and the number of tasks

Fig. 9: Performance of various evaluation metrics of different methods with job submission rate 50 jobs/sec and 30 workers on Amazon EC2.

ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grant NSF-2400459 and the generous funding from the Cyber Policy Institute at Florida A&M University. We would like to thank Mr. Sunday J. Awine, Kalab M. Kiros, Javonte L. Carter, and Ms. Lam Phuong Nguyen for their help on this work.

REFERENCES

[1] 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 *Proc. of SoCC*, San Jose, 2012.

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

[3] T. Jin, Z. Cai, B. Li, C. Zheng, G. Jiang, and J. Cheng. Improving resource utilization by timely fine-grained scheduling. In *EuroSys*, 2020.

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

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

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

[7] J. Rasley, K. Karanasosy, S. Kandulay, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *EuroSys*, 2016.

[8] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *NSDI*, 2022.

[9] J. Liu, Y. Lao, Y. Mao, and R. Buyya. Sailfish: A dependency-aware and resource efficient scheduling for low latency in clouds. In *Proc. of IEEE Big Data*, 2023.

[10] M. Hosseini, M. Salehi, and R. Gottumukkala. Enabling interactive video streaming for public safety monitoring through batch scheduling. In *Proc. of HPCC/SmartCity/DSS*, 2017.

[11] R. Damaševičius, N. Bacanin, and S. Misra. From sensors to safety: Internet of emergency services (ioes) for emergency response and disaster management. *J. Sens. Actuator Netw.*, 12(3):41, 2023.

[12] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[13] M. Harchol-Balter. Task assignment with unknown duration. In *Proc. of ICDCS*, 2000.

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

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

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

[17] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of SoCC*, 2012.

[18] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. of SoCC*, 2013.

[19] A. Qiao, S. K. Choe, J. S. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proc. of OSDI*, 2021.

[20] J. Gu, M. Chowdhury, K. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *Proc. of NSDI*, 2019.

[21] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *Proc. of EuroSys*, 2020.

[22] S. Vijayakumar, A. Madhavapeddy, and E. Kalyvianaki. Scheduling for reduced tail task latencies in highly utilized datacenters. In *SoCC*, 2024.

[23] W. Xiang, Y. Li, Y. Ren, F. Jiang, C. Xin, V. Gupta, C. Xiang, X. Song, M. Liu, B. Li, K. Shao, C. Xu, W. Shao, Y. Fu, W. Wang, C. Xu, W. Xu, C. Lin, R. Shi, and Y. Liang. GÖDEL: Unified large-scale resource management and scheduling at bytedance. In *SoCC*, 2023.

[24] T. T. Chang and S. Venkataraman. Eva: Cost-efficient cloud-based cluster scheduling. In *Proc. of EuroSys*, Rotterdam, 2025.

[25] E. Guo, W. Jia, X. Ding, and J. Shan. Optimizing task scheduling in cloud vms with accurate vcpu abstraction. In *Proc. of EuroSys*, 2025.

[26] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S. Chung, and Y. Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Comput. Surv.*, 47(4):1–33, 2015.

[27] B. Wu, K. Qian, B. Li, Y. Ma, Q. Zhang, Z. Jiang, J. Zhao, D. Cai, E. Zhai, X. Liu, and X. Jin. Xron: A hybrid elastic cloud overlay network for video conferencing at planetary scale. In *SIGCOMM*, 2023.

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

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

[30] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang. Beware of fragmentation: Scheduling gpu-sharing workloads with fragmentation gradient descent. In *NSDI*, 2023.

[31] L. Chen and H. Shen. Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters. In *Proc. of INFOCOM*, 2017.

[32] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: Vm allocation service at scale. In *Proc. of OSDI*, 2020.

[33] Amazon ec2. http://aws.amazon.com/ec2 [accessed in May 2025].

[34] Palmetto cluster. https://docs.rcd.clemson.edu/palmetto/about/ [accessed in Mar. 2025].

[35] A. Balasubramanian, A. Sussman, and N. Sadeh. Decentralized preemptive scheduling across heterogeneous multi-core grid resources. In *JSSPP*, 2013.

[36] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. on Computer Systems*, 21(2):207–233, 2003.

[37] Google trace. https://code.google.com/p/googleclusterdata/ [accessed in May 2025].