Online Multi-Resource Scheduling for Minimum Task Completion Time in Cloud Servers

MohammadJavad NoroozOliaee*, Bechir Hamdaoui*, Mohsen Guizani#, and Mahdi Ben Ghorbel#
*Oregon State University, Oregon, USA | Emails: noroozom,hamdaoub@onid.orst.edu
#Qatar University, Doha, Qatar | Emails: mahdi.benghorbel,mguizani@qu.edu.qa

***Abstract*—We design a simple and efficient online scheme for scheduling cloud tasks requesting multiple resources, such as CPU and memory. The proposed scheme reduces the queuing delay of the cloud tasks by accounting for their execution time lengths. We also derive bounds on the average queuing delays, and evaluate the performance of our proposed scheme and compare it with those achievable under existing schemes by relying on real Google data traces. Using this data, we show that our scheme outperforms the other schemes in terms of resource utilizations as well as average task queuing delays.**

I. Introduction

Demand for cloud-based services has been increasing in recent years. As a result, cloud computing has been attracting the focus and attention of many researchers to address numerous related issues, such as energy reduction in data centers, online job scheduling, realtime scheduling, etc. There have also been some research efforts aimed to develop service models and architectures suitable for cloud computing paradigm [1, 2]. Some other works try to adapt and apply existing solutions developed for similar problems to solve cloud computing problems. Examples of such problems are bin packing, vector bin packing, off-line scheduling, online scheduling, and fair resource allocation [3–7].

In [3], the authors propose an online scheduling algorithm for real-time cloud computing services using two utility functions for rewarding early completion of tasks and penalizing missed deadlines. In [8], the authors consider a market-based resource allocation model for batch jobs in cloud computing and propose an approximation algorithm in order to maximize social welfare. In [5], join-shortest-queue and MaxWeight scheduling policies and power-of-two-choices routing are proved to be optimal under the proposed model. In [9], a genetic algorithm is improved by combining the Min-Min and Max-Min concepts to schedule independent tasks in cloud computing. Another genetic algorithm is also proposed in [10], which guarantees the best solution in finite time for cost-based multi QoS scheduling in cloud computing. In [11], the authors present a fully decentralized scheduler that benefits from information aggregation to allocate available nodes to a task so that it finishes in time. They show that their proposed scheduler has competitive performance in networks of up to hundred thousand nodes. In [12], the authors minimize total energy cost while trying to meet as many client requests as possible. The system is penalized if the service time of a task exceeds a specific upper bound. In [13], the authors propose an efficient multi-objective scheduling algorithm for workflow applications, where the objectives to minimize include economical, energy, and reliability considerations. Ramamritham et al. [14] were among the first to propose scheduling algorithms for tasks with both time and resource limitations. The authors in [4] consider different heuristics based on task demands using offline scheduling algorithms.

Most of the mentioned works consider heuristics based on demand values or task completion deadlines. Instead, we aim, in this work, to minimize the average task queuing delay while accounting for task execution times. For this, we propose a simple and efficient online scheme for scheduling cloud tasks that request multiple resources. We derive bounds on the average queuing delays, and evaluate the performance of our proposed scheme and compare it with those achievable under existing schemes by relying on real Google data traces [15]. Using this real Google data, we show that our scheme outperforms the other schemes in terms of resource utilizations as well as average task queuing delays.

The paper is organized as follows. In Section II, we introduce some terminology. Section III provides some background and state our motivation and objective. Section IV presents our proposed scheme. In Section V, we provide upper and lower bounds on average task queuing delays. Section VI presents the performance evaluation of our scheme. We conclude our work in Section VII.

II. Terminology and Notation

In this section, we describe the different system components essential to the scheduling problem that we investigate in this paper.

Servers: They are the computing resources that run the tasks submitted by cloud clients. Upon its arrival, a task is assigned to a server for execution. In our problem, we assume that the system contains n heterogeneous servers each with two types of resources, a processing unit (CPU) and a memory unit (RAM). For simplicity and without loss of generality, we consider throughout this paper the normalized capacity of the resource with respect to the maximum available capacity among all servers.

Task: It is the entity that is to be submitted by cloud clients, needs to assigned to and executed on server. Each task requests an amount of CPU and Memory resources to be allocated to it for a specific period of time. Each task i is represented by a 4-tuple as $(t_i^s, t_i^{exec}, c_i, m_i)$ where: t_i^s and t_i^{exec} denote the task's submission and execution times, and c_i and m_i denote respectively task i's requested amounts of cpu and $memory$. Without loss of generality, we

assume that c_i and m_i are also normalized with respect to the maximum resource capacity among all available servers. That is, $0 \leq c_i \leq 1$ and $0 \leq m_i \leq 1$ for every task i. We also assume that c_i and m_i are the real values of task i demands (i.e. the tasks reveal their true values of their demands). However, one may argue how one can ensure that these values are real. For this, we assume that either the tasks are internal, meaning that no outsider is using the system, or when there are malicious tasks, there is a task admission control (TAC) mechanism that only allows the submission of non-malicious tasks. TAC mechanisms are beyond the scope of this work.

Scheduler: In this work we focus on designing this component. The scheduler determines when and on which server each task starts to execute. However, the scheduler considers the constraint that the total demands for cpu and $memory$ of the tasks that run on a server at the same time must not exceed the servers available cpu and $memory$, respectively.

Queue: It holds the tasks that are submitted, but have not started their execution yet. Upon a task's arrival/submission, if the scheduler does not find a idle server to which the task can be assigned for execution, it queues the task. The amount of time a task spends in the queue is called queuing delay. Queuing delay can be zero if the task starts running right after submission.

In this work, our aim is to design a scheduler that reduces the average queuing delay of the tasks under the constraints mentioned above. In Section IV, we develop a simple scheduler that is based on existing schemes and takes advantage of a new heuristic based on the tasks' execution times.

III. Background, Motivation and Objective

In this section, we overview some existing schemes that are used mostly for task scheduling in cloud computing. Then, we state our motivation as well as objective for this work.

A. Existing Schedulers

The problem of reducing the average queuing delay of tasks can be also thought of as a problem of reducing the number of servers to be used for task execution. Intuitively, using fewer servers results in shorter queuing delays, since this means that there are more servers (resources) left available for running other tasks (already submitted or to be submitted in the future). Clearly, the more resources/servers we have, the higher the chances that a submitted task gets executed fast, which in turn results in reducing the task's queuing delay.

Our problem is now converted to a classical bin packing problem in which it is assumed that there are items (e.g., tasks) that need to be placed in the minimum possible number of bins (e.g., servers) subject to bin capacity constraints; i.e., none of the bins should contain items whose total capacity exceeds the bin's capacity. However, the bin packing problem considers that the items each has a single size value, which corresponds to the item's size. In our problem, each item or task is associated with two values, once represents the CPU demand and one represents the memory demand. Vector bin packing problem is then more suitable to our scheduling problem, which is a generalization of classic bin packing problem to allow items to have multiple values. In vector bin packing problems, each item is considered as a d-dimensional vector that should be placed in d-dimensional bins. Our resource scheduling problem is then a 2-dimensional vector bin packing. In what follows, we overview few existing scheduling schemes used for solving vector bin packing that are relevant/applicable to our studied problem.

1) Online scheduling: We first begin by overviewing some online task scheduling algorithms. In online scheduling, it is assumed that tasks can be submitted at different times, and the schedule is to assign them to servers as they arrive.

First Fit (FF): In this algorithm, when a task is submitted, the scheduler looks through all the servers one by one, and assigns the task to any available server it finds. But if there is no available server with enough resources to accommodate the task, the task is assigned to a new server; i.e., the number of ON servers increases by one as a result of this assignment. In our scheduling problem, the number of available servers is limited. Hence, in order to apply FF algorithm, we modify it as follows. When there is no available servers with enough capacity to fit the new task, the scheduler appends the task to the end of the queue.

When the execution of a task completes, the scheduler checks whether it can assign the task at the front of the queue to the corresponding server. If the server has enough resources, the task is assigned to it; otherwise the scheduler waits for the next task submission or completion.

Best Fit (BF): This algorithm is similar to FF algorithm. The only difference is that the scheduler tries to find a server that has the least amount of remaining resource after the task is assigned to it. Note that in the case of the d-dimensional problem, the least amount of remaining resource has no trivial definition, since the size of a task is not a single value any longer. Later, we will see some heuristics that define the size of multi-dimensional tasks, which can also be used to define the amount of remaining resource of a server.

Worst Fit: This algorithm is similar to BF, except the scheduler tries to find the server that has the largest amount of remaining resources (instead of the least amount of resources as in BF) after the task is assigned to it. Again, like in BF algorithm, the definition of the amount of remaining resource requires size definition of multi-dimensional tasks.

Random Fit: As the name suggests, the scheduler randomly assigns the task to any available server with enough capacity; i.e., without violating the capacity constraint.

It is known that the BF algorithm outperforms the other algorithms in most cases, and that is why it is widely used for resource scheduling in cloud computing.

2) Offline scheduling: We now describe some existing offline scheduling algorithms. In these algorithms, it is assumed that the scheduler knows all the tasks that are to be scheduled ahead of time; for example, this applies to when all tasks arrive (are submitted) at the same time.

First Fit Decreasing (FFD): In this algorithm, the scheduler sorts all tasks in descending order based on some metric specified by the heuristic used by the algorithm, and then

assigns the tasks according to the FF algorithm.

Best Fit Decreasing (BFD): The scheduling algorithm is similar to FFD, except the task assignment is based on the least amount of remaining resource as it is described above.

3) Multi-dimensional task size metrics: We now present some heuristic metrics used to define and specify the sizes of tasks with multiple dimension. The metrics presented next are proposed [4].

Product: The size of a multi-dimensional task is calculated by multiplying all demands. In our problem, the Product size of task i equals $c_i \times m_i$.

Sum: The size of each task is defined as a linear combination of all demands, where the constants (weights) indicate the importance of the resources. In our problem, the size of task i equals $a_1 c_i + a_2 m_i$ where a_1 and a_2 are weights reflecting the importance of cpu and $memory$, respectively.

AvgSum: This is the same as the "Sum" metric above, except the weights are the resource's average demands. For our problem, we have $a_1 = 1/k \sum_{i=1}^{k} c_i$ and $a_2 = 1/k \sum_{i=1}^{k} m_i$ where k is the total number of submitted tasks.

DotProduct (DP): This corresponds to the dot product between the demands of the task and the remaining amount of resource. Thus, each task does not have a single value, but rather it has a different value for each server. Then, the maximum value over the servers is used to determine which server the task should be assigned to without violating the capacity constraint.

Norm-Based Greedy (NB): This defines the size of each task as its L_p-norm value with $p > 1$. Consider a vector $\vec{x} = (x_1, ..., x_d)$, then L_p norm of $\vec{x}$ is equal to

$$\left(\sum_{j=1}^{d} |x_j|^p \right)^{1/p}$$

In our problem, L_2 norm of task i is equal to $\sqrt{c_i^2 + m_i^2}$.

As shown in [4], Dot-Product and L_2 Norm-based Greedy heuristics outperform other heuristics in most cases.

B. Motivation and Objective

As explained earlier in this section, there have been proposed many heuristics and algorithms for online task scheduling. Most of these heuristics consider task demands only, and ignore task execution times. Nevertheless, we know that in order to minimize the queuing delay of the tasks when all tasks have the same size, we need to execute the task with the shortest execution time. Unlike previous works, this paper considers and accounts for task execution times when sorting tasks for execution. The objective of this work is then to design a scheduling heuristic with an objective of reducing the average queuing delay of the submitted tasks while accounting for task execution times. This problem is interesting, since developing a good algorithm is neither easy nor enough to address all issues raised in scheduling. In this work, we propose to build a two-phase scheduling scheme for multi-dimensional tasks, using the idea of a well-known greedy algorithm for off-line task scheduling (with only one dimension) in which the scheduler sorts the task based on their execution time in ascending order and runs the task with the shortest execution time. This greedy algorithm is optimal and can be implemented using a simple and efficient sorting algorithm such as quick-sort.

IV. The Proposed Scheduling Scheme

ïż£ In this section, we explain our proposed scheduling scheme. Our scheme schedules tasks in two phases. Phase one is triggered by an arrival of a new task, whereas phase two is triggered by the completion of a task execution.

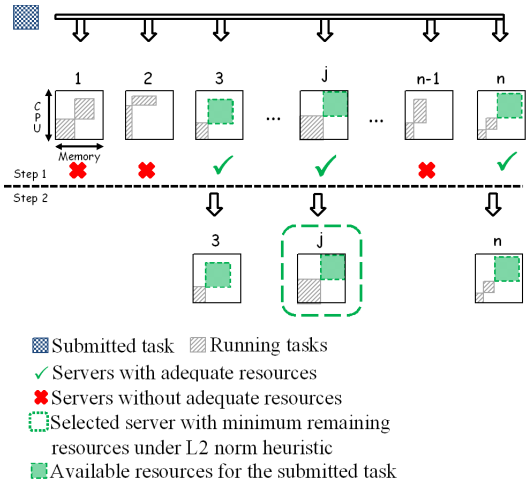
The first phase is triggered by a new task arrival event. Upon the arrival of a new task, the scheduler tries to find an available server for assigning and executing this recently submitted task. If the scheduler does not find any server for the task, it puts it in the queue. Specifically, in the first phase, the task is assigned to the server with the minimum L_2-norm of the remaining resources, given the server has sufficient resources to accommodate the task. If there is no server that satisfies these conditions, the task is placed in the queue sorted according to its execution time.

Fig. 1(a) illustrates how the scheduling is conducted in the first phase. As shown in the figure, in case of a new task arrival event, there are two steps the scheduler performs: in step 1, the scheduler filters out the servers that have adequate amounts of resources to allocate to the task; and in step 2, the scheduler finds the server with the minimum L_2 norm of the remaining resources. When no server is found in step 1, the task will be queued as explained before.

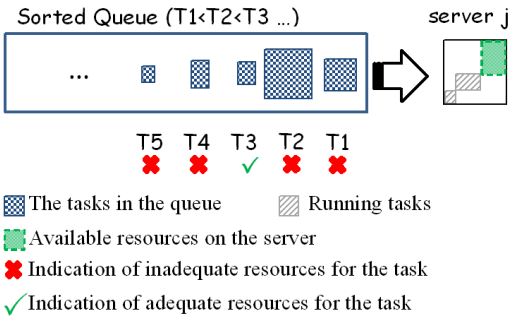
In the second phase which is triggered upon a task execution completion, the scheduler tries to find the task with the shortest execution time among the tasks that are already in the queue, and fits it on the recently released server. Since the queue is kept sorted according to task execution times, finding the task with the shortest execution time is done by iterating over all the tasks in the queue. Then, the tasks that fit the server will be assigned to it after allocating resources to all previous tasks with shorter execution times.

Fig. 1(b) illustrates how scheduling benefits from the proposed heuristic during the second phase when a task running on server j completes. The figure shows that the first two tasks in the queue need more resources than what it is available on the recently released server j, but the third task in the queue requires less resources. Hence, server j allocates its remaining resources to this third task. It is seen that, even though the fifth task in the queue requires less resources than the third task, but there is not enough resources to allocate to the fifth after allocating resources to the third task.

Our proposed heuristic is inspired by the well-known greedy scheduling algorithm described in Section III. However, the scheduler uses this heuristic in its second phase only and chooses the task with the shortest execution time among the tasks which does not violate the corresponding server's capacity constraints. We call, throughout the paper, our proposed scheme BF-EXEC, since in its first phase, it uses the BF algorithm and in its second phase, it schedules tasks according to their execution times.



(a) Phase 1: Scheduling upon task arrival



(b) Phase 2: Scheduling upon execution completion

Fig. 1. Scheduling Scheme: (a) A task is submitted and the scheduler finds the best server (b) Scheduling the task with the shortest execution time in the queue upon a task execution completion

V. Average Queuing Delay Bounds

We derive upper and lower bounds on the average queuing delay. In what follows, we assume that the tasks' average arrival rate and service time are λ and μ, respectively.

Theorem 5.1: If $c_i \leq U_c$ and $m_i \leq U_m$ for every task i, the queuing delay, W, is upper bounded as

$$W \leq \mu - \frac{1}{\lambda}\lfloor max\{\frac{1}{U_c}, \frac{1}{U_m}\}\rfloor$$

Proof: We prove it by assuming that all the tasks are identical in terms of their demands. In order to find an upper bound, we assume $c_i = U_c$ and $m_i = U_m$ for every task i. Hence, we know that the number of running tasks N_r, is bounded as $N_r \geq \lfloor max\{\frac{1}{U_c}, \frac{1}{U_m}\}\rfloor$.

Recall that the number of tasks in the system, N, is, as given by Little's Theorem, $N = \lambda\mu$. As a result, the number of tasks waiting in the queue is $N_q = N - N_r$ which yields $N_q \leq \lambda\mu - \lfloor max\{\frac{1}{U_c}, \frac{1}{U_m}\}\rfloor$. Using Little's theorem for the queue system this time, we have $N_q = W/\lambda$. It then follows the upper bound on the queuing delay as stated in the theorem. ■

Theorem 5.2: If $L_c \leq c_i$ and $L_m \leq m_i$ for every task i, the queuing delay, W, is lower bounded as follows

$$W \geq \mu - \frac{1}{\lambda}\lfloor min\{\frac{1}{L_c}, \frac{1}{L_m}\}\rfloor$$

Proof: Similarly, we prove this by assuming that all the tasks are identical in terms of their demands, and considering the worst case scenario, where $c_i = L_c$ and $m_i = L_m$ for every task i. Hence, the number of running tasks, N_r, is upper bounded by $\lfloor min\{\frac{1}{L_c}, \frac{1}{L_m}\}\rfloor$. Little's theorem yields that the number of tasks in the system, $N = \lambda\mu$. Hence, the number of tasks waiting in the queue is $N_q = N - N_r$ which yields $N_q \geq \lambda\mu - \lfloor min\{\frac{1}{L_c}, \frac{1}{L_m}\}\rfloor$. Using Little's theorem for the queue this time, we have $N_q = W/\lambda$. The lower bound stated in the theorem follows then. ■

VI. Performance Evaluation

ïż£In this section, we evaluate our proposed techniques by using real Google data traces [15]. Using this real data, we show that our proposed scheme performs well when compared to existing schemes. For our evaluation, we extract the arrival time and execution time along with cpu and $memory$ demands of 2000 tasks from the data traces provided by Google. Google provides cpu and $memory$ demands of the tasks in the normalized form as mentioned in Section II. We will compare our scheme to the following schemes: FFD-NB (First Fit Decreasing with L2 Norm-Based), FFD-DP (First Fit Decreasing with Dot-Product), and FCFS (First-Come First-Serve)[1].

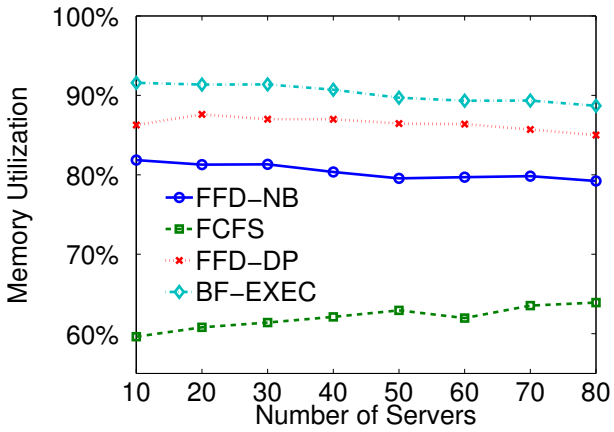
A. Resource Utilization

Fig. 2 depicts the average CPU (Fig. 2(b)) and memory (Fig. 2(a)) resource utilizations while varying the number of servers from 10 to 80 under the different scheduling schemes. Note that our proposed scheme utilizes almost 90% of the servers' memory capacities which is higher than other schemes' utilizations. Also, it can be seen from Fig. 2(b) that our proposed scheme utilizes 85% to 90% of the servers' cpu capacities which is higher than other schemes' CPU utilizations. Although higher utilization means that lower resource wastage, it does not mean that system resources are utilized in favor of reducing the average task queuing delay. This will be studied in next section.

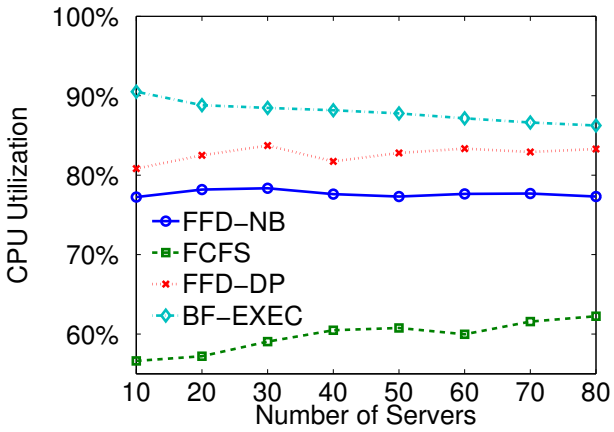
B. Average Delay

Fig. 3 depicts the average task queuing delay for various numbers of servers under the different studied scheduling schemes. Note that our proposed scheme reduces the average queuing delay when compared to the other schemes. We conclude that our proposed scheme utilizes the resources effectively while reducing the average queueing delay that tasks experience.

[1]**As the name suggests, in this scheme, a task must be assigned to a server before any other task that arrives after it can be assigned. Here, if no server with sufficient resources is found to run a given task, then all the tasks that came after it must wait until a server is found to execute this given task.**



(a) Memory Utilization



(b) CPU Utilization

Fig. 2. Resource utilization for various number of servers under different scheduling schemes

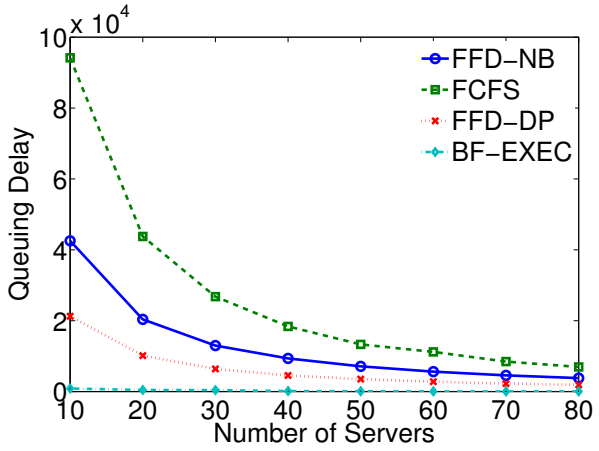


Fig. 3. Average queuing delay for various number of servers under different scheduling schemes

VII. Conclusion

This paper proposes an online scheduling scheme that aims to minimize the average task queuing delay while accounting for task execution times. Our focus is on scheduling tasks that request multiple resources, such as CPU and memory. We use real Google data traces to evaluate the performance of our proposed scheme and compare it with those achievable under some existing ones. Our results show that our scheme outperforms the other schemes in terms of resource utilizations as well as average task queuing delays.

VIII. Acknowledgment

This work was made possible by NPRP grant # NPRP 5-319-2-121 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] Ravi Shankar Dhakar, Amit Gupta, and Ashish Vijay, "Cloud computing architecture," .

[2] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 2011.

[3] Shuo Liu, Gang Quan, and Shangping Ren, "On-line scheduling of real-time services for cloud computing," in *Services (SERVICES-1), 2010 6th World Congress on*, 2010, pp. 459–464.

[4] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder, "Heuristics for vector bin packing," 2011.

[5] S.T. Maguluri, R. Srikant, and Lei Ying, "Heavy traffic optimal resource allocation algorithms for cloud computing clusters," in *Teletraffic Congress (ITC 24), 2012 24th International*, 2012, pp. 1–8.

[6] Leah Epstein and Rob van Stee, "Optimal online algorithms for multidimensional packing problems," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 431–448, 2005.

[7] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica, "Multi-resource fair queueing for packet processing," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 1–12, 2012.

[8] Navendu Jain, Ishai Menache, Joseph Naor, and Jonathan Yaniv, "Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters," in *of the 24th ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 255–266.

[9] Pardeep Kumar and Amandeep Verma, "Scheduling using improved genetic algorithm in cloud computing for independent tasks," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*. 2012, ICACCI '12, pp. 137–142, ACM.

[10] D. Dutta and R. C. Joshi, "A genetic: algorithm approach to cost-based multi-qos job scheduling in cloud computing environment," in *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. 2011, ICWET '11, pp. 422–427, ACM.

[11] J. Celaya and U. Arronategui, "A highly scalable decentralized scheduler of tasks with deadlines," in *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, 2011, pp. 58–65.

[12] H. Goudarzi, M. Ghasemazar, and M. Pedram, "Sla-based optimization of power and migration cost in cloud computing," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012, pp. 172–179.

[13] H.M. Fard, R. Prodan, J.J.D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012, pp. 300–309.

[14] K. Ramamritham, J.A. Stankovic, and Wei Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *Computers, IEEE Transactions on*, vol. 38, no. 8, pp. 1110–1123, 1989.

[15] "Google cluster data, https://code.google.com/p/googleclusterdata/," 2011.