Minimising the Execution of Unknown Bag-of-Task Jobs with Deadlines on the Cloud

Long Thai
School of Computer Science
University of St Andrews
Fife, Scotland, UK
ltt2@st-andrews.ac.uk

Blesson Varghese
School of EEECS
Queen's University Belfast
Belfast, N. Ireland, UK
varghese@qub.ac.uk

Adam Barker
School of Computer Science
University of St Andrews
Fife, Scotland, UK
adam.barker@st-andrews.ac.uk

ABSTRACT

Scheduling jobs with deadlines, each of which defines the latest time that a job must be completed, can be challenging on the cloud due to incurred costs and unpredictable performance. This problem is further complicated when there is not enough information to effectively schedule a job such that its deadline is satisfied, and the cost is minimised. In this paper, we present an approach to schedule jobs, whose performance are unknown before execution, with deadlines on the cloud. By performing a sampling phase to collect the necessary information about those jobs, our approach delivers the scheduling decision within 10% cost and 16% violation rate when compared to the ideal setting, which has complete knowledge about each of the jobs from the beginning. It is noted that our proposed algorithm outperforms existing approaches, which use a fixed amount of resources by reducing the violation cost by at least two times.

Keywords

Bag-of-Task, Scheduling, Deadline, Cloud computing, Unknown

1. INTRODUCTION

Nowadays, cloud computing, especially Infrastructure as a Service (IaaS), is widely used by organisations due to its flexible resource on demand model. Instead of building a data centre which requires a huge upfront investment, an organisation can simply acquire and pay for virtual resources. Moreover, cloud computing also removes the cost and time overhead for managing and maintaining physical resources.

On the other hand, adopting cloud computing can be challenging because of its unique characteristics. First of all, due to its pay-as-you-go scheme, any decision regarding using the cloud resources has to take monetary cost into account, as a user is billed as soon as the resources are acquired.

Secondly, in order to satisfy the diversity of requirements of different organisations, cloud providers often offer a wide variety of machine types which are different from each other in not only hardware configuration but also price. On the contrary, none of the providers offer any support for a user to determine, which type or combination of multiple types is suitable for running application(s) so that the desired performance is achieved while remaining cost effective.

Thirdly, the performance of cloud resources is not absolutely guaranteed. For instance, Ward and Barker reported that the performance of different virtual machines (VMs) of the same type provided by Amazon Web Service (AWS) [1] could be vary up to 29% [20]. Moreover, when a user requests for cloud resources, there is a waiting time before the resources become available. Mao and Humphrey reported that this wait time could be some times more than 800 seconds [13]. As a result, a user needs a mechanism to handle the performance uncertainty during runtime.

The last challenge is not directly from cloud computing but from the need to ensure the Quality-of-Service (QoS) of applications. According to Microsoft, most of its jobs need to finish within predefined deadlines [7]. Satisfying QoS can be more difficult for applications running on the cloud due the three challenges discussed above.

In this paper, we present our work in scheduling the execution of multiple jobs with deadlines on the cloud so that the monetary cost can be kept minimal. Our research focuses on Bag-of-Tasks (BoT) job which contains multiple independent tasks and are widely used by both scientific communities [12] and industrial organisations [9]. In order to perform scheduling, it is necessary to know the *task execution times*, i.e. how long it takes to execute a single task of a job on a VM of any type. However, this kind of knowledge may not be available for unknown jobs which have never been seen before.

The main contribution of this paper is an approach which schedules unknown jobs with deadlines on the cloud by sampling in order to retrieve a job's characteristics which can be used for scheduling. Our proposed approach is a continuous mechanism which is able to handle jobs which are submitted at different times.

Based on our evaluation, the proposed approach is able to schedule jobs without prior knowledge and keep the violation cost within 30% compared to the ideal case when there is full knowledge of the application. Moreover, our approach outperforms other ones which use fixed amount of resources by keeping the violation cost at least two times lower.

This paper is structured as follows. Section 2 presents the related work and distinguishes our research. Section 3 in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

troduces the mechanism for scheduling jobs with deadlines when task execution times are already available. Section 4 proposes an approach to handle an unknown job. Section 5 introduces a dynamic reassignment feature to handle performance uncertainty during runtime. Our approach is evaluated in Section 6. Section 7 concludes this paper.

2. RELATED WORK

Scheduling job executions within a distributed environment has always received a lot of attention by both academic and industrial researchers. The academic community mostly focuses on Grid computing due to its nature which facilitates the collaboration between many organisations for sharing computing resources [8]. Each proposed research aims to optimise one or multiple criterion regarding the performance of an application on the grid. Ranganathan and Foster proposed the task assignment and data replication approaches to reduce the geographical distance between data and computation [17]. Beaumont et al. aimed to maximise the number of tasks executed concurrently while ensuring fair resource sharing between applications [3]. Benoit et al. took another approach to minimise the application runtime while sharing resources betweens applications [4].

On the other hand, researchers working in industry seem to be more interested in scheduling applications and jobs in cluster environment since their organisations can afford to build private data centres. In general, those scheduling frameworks aim to minimise the runtime of jobs executed within a data centre [9, 5]. Furthermore, Microsoft has reported that their scheduling framework takes job deadlines into account while perform scheduling [7]. Academic researchers have developed frameworks, such as Sparrow [16] and Mesos [11], with the goal of minimising the execution time of jobs on the cluster.

When scheduling jobs on grid and cluster environments, the goal is to greedily acquire existing resources as much as possible and as quickly as possible. However, this strategy cannot be applied on cloud environment due to the monetary cost involved for acquiring and using resources. As a result, the scheduling mechanism must take into account not only performance but also monetary cost.

Amato et al. [2] and Garcia et al. [10] proposed cloud brokering approaches to minimise cost while satisfying requirements in term of hardware resources, such as the number of CPU cores or the amount of memory. However, these approaches may not be applicable since the same amount of cloud hardware can have different performance at different times, as mentioned in the previous section.

More recent researchers, including ourselves, have started to investigate the problem of scheduling BoT jobs on the cloud with constraints such as budget or deadlines based on task execution times which were assumed to be known [14, 19, 18]. The work of Oprescu et al. addresses a similar problem reported in this paper by proposing an approach to schedule a BoT job on the cloud without any knowledge regarding task execution times [15]. Nevertheless, our research can be distinguished from any existing research in the following two ways: (i) our approach is able to handle multiple jobs, and (ii) we ensure QoS by satisfying job deadlines instead of monetary budget.

3. SCHEDULING JOBS WITH DEADLINES ON THE CLOUD

In this section, we introduce an approach to schedule a job execution on the cloud so that not only is its deadline satisfied but also the monetary cost is minimised.

3.1 Resource Selection Model

Let IT denote all available instance types. Each instance type $it \in IT$ has a price p_{it}. It should be noted that each instance type is charged **per time block** which contains Θ seconds, for instance a one hour time block contains 3600 seconds. In other words, a user still has to pay for the full time block even if the VM runs for less than one hour.

Let j be the submitted job that needs to be scheduled. It belongs to an application a_j and its number of tasks is n_j. We separate the job and application so that it is possible for jobs of the same application to be submitted more than one time, e.g. recurring jobs. Finally, all of its tasks must be completely executed by the deadline d_j.

Let $e_{j,it} = e_{a_j,it}$ be the average task execution time of a job j (or an application a_j) on an instance of type it. Which means that $e_{j,it}$ is the amount of time in seconds that it takes for an instance of it to finish executing a task of job j. Notably, $e_{j,it}$ is just an average value, the actual task execution time can vary in runtime. This issue will be discussed and handled in Section 5.

Assume that a job is submitted at the time $\Gamma_0 = 0$, the amount of time from when a job is submitted to its deadline is $d_j - \Gamma_0 = d_j - 0 = d_j$. Notably, when a new instance is created, it is not ready immediately but takes a small but noticeable amount of time to be ready, this overhead is denoted as β. As the result, the actual available time to execute a job within its deadline can be calculated as $d_j - \beta$.

The number of tasks of a job j that can be executed by one instance of type it can be calculated by dividing the available execution time to the time it takes to execute on task since an instance can execute only one task at a time:

$$n_{j,it} = \lfloor \frac{d_j - \beta}{e_{j,it}} \rfloor \tag{1}$$

The floor function is applied as a task must be fully executed.

Assuming that $ni_{j,it}$ is the number of instances of type it which are created in other to execute tasks of j. Hence, the total number of tasks of j executed by instances of type it is $ni_{j,it} \times n_{j,it}$. As the result, the total number of tasks of j executed by all instances of all types can be calculated as:

$$\sum_{it \in IT} (ni_{j,it} \times n_{j,it}) \tag{2}$$

Since all tasks of a job must be executed, the total number of executed tasks must be equal to or greater than a job's number of tasks.

$$n_j \leq \sum_{it \in IT} (ni_{j,it} \times n_{j,it}) \tag{3}$$

The constraint presented by Equation 3 ensures that all tasks are executed within a job's deadline.

In one time block, the total cost can be calculated as:

$$\sum_{it \in IT} ni_{j,it} \times p_{it} \quad (4)$$

An instance's running time is from it is created to the time it finishes the last tasks. As each instance has to finish its execution before a job deadline d_j, it is reasonable to assume that an instance's running time is equal to the deadline. The number of time blocks used by one instance in order to execute a job is $\lceil \frac{d_j}{\Theta} \rceil$. The ceiling function is used in order to round up any fraction of a time block to a full time block.

Assuming that workload is evenly distributed among all instances, hence, all of them finish execution nearly at the same time. In other words, their execution times are nearly identical. The total cost of executing a job within its deadline can be calculated by multiplying the number of used time blocks to the cost of each time block:

$$\lceil \frac{d_j}{\Theta} \rceil \times \sum_{it \in IT} (ni_{j,it} \times p_{it}) \quad (5)$$

As the result, the problem of scheduling a job so that its deadline is not exceeded and the cost is minimised can be modelled as the following linear programming problem:

$$\begin{aligned} \text{minimise} \quad & \lceil \frac{d_j}{\Theta} \rceil \times \sum_{it \in IT} (ni_{j,it} \times p_{it}) \\ \text{subject to} \quad & n_j \leq \sum_{it \in IT} (ni_{j,it} \times n_{j,it}) \\ & n_{j,it} = \lfloor \frac{d_j - \beta}{e_{j,it}} \rfloor \end{aligned} \quad (6)$$

3.2 Continuous Job Scheduling

It is assumed that when a job is submitted, it is put into a first-in-first-out queue and waits to be scheduled. In other words, only one job is scheduled at a time. Moreover, when a job is scheduled, there could be some existing instances which are executing jobs submitted in the past. Hence, it is possible to re-use those existing instances to execute tasks of a new job, as long as it does not result in any deadline violation or additional cost. In the best case scenario, all tasks from a new job are assigned to the existing instances and no new instance is required, which means no additional cost as well.

When it is not possible to assign all tasks of a new job to the existing instances, new instances must be created to handle the rest. The number of new instances of each type can be calculated by solving the Model 6.

Algorithm 1 Jobs Scheduling

```
1: function SCHEDULE(I, J)
2:     for j ∈ J do
3:         Assign tasks of j to I
4:         if there are remaining tasks then
5:             I_j ← the result of solving Model 6
6:             I ← I ∪ I_j
7:         end if
8:     end for
9: end function
```

The scheduling progress is presented in Algorithm 1. Its inputs are the list of submitted jobs (J) and the list of existing instances (I).

For each submitted job, Algorithm 1 tries to assign its tasks to existing instances (Line 4). Assuming that each instance maintains a queue of tasks waiting to be executed, this assignment process simply adds tasks of a submitted job to the end of that queue as long as the task can be fully executed before its job's deadline.

After assigning tasks to existing instances, if there are remaining tasks, it is necessary to create new instances. The number of instances of each type can be calculated by solving Model 6 (Line 5). It should be noted that only remaining tasks are taken into account.

Finally, the new instances which are created for a submitted job are added to the list of existing ones (Line 6).

4. HANDLE UNKNOWN JOB

In Section 3, it is assumed that the task execution times of a job (or its application) on all instance types are known prior to its submission. In this section, we present an approach for scheduling a job with deadline on the cloud when its task execution times are unknown. The main idea is to split an execution into two phases:

- **Sampling phase**: in which some tasks of a jobs are executed on instances of all available types so that their actual execution times can be retrieved. The task execution time of a job on an instance type is calculated as the mean of all actual execution times of all sampling tasks assigned to an instance of that type.
- **Full execution phase**: based on the average task execution times, the remaining tasks of a submitted job are scheduled to be executing using an approach presented in Section 3.

4.1 Determine the Sampling Duration

When a job of an unknown application is submitted, the sampling phase starts. In this phase, some tasks of a job are assigned to an instance of each type for execution. The actual execution time of an task is retrieved and later used to estimate the average task execution of a job's application on each instance type.

The most important factor to consider before running the sampling phase is its duration, i.e. how long the sampling phase should last. If a duration is too short, only few, or even none, of the sampling tasks are executed, hence the retrieved data may not be sufficient to estimate the average task execution time. On the other hand, if a duration is too long, the full execution phase is delayed further, which means the available time for execution is shortened. Notably, while calculating this duration, the instance creation overhead must be taken in to account since it may require to create new instances in both the sampling duration, when an instance type has no available instance to receive sampling tasks, and the full execution phase, when new instances are required to execute a job within its deadline.

In our approach, a duration of the sampling phase is calculated as a fraction of a job's available execution time, the amount of time from when a job it submitted to its deadline. For instance, the sampling phase can take 5% or 10% of a job's available execution time. As a result, we assume that when a job of an unknown application is submitted,

its available execution time must be large enough to have a sufficient amount of time for a duration of the sampling phase. The affect of a length of a duration of the sampling phase will be investigated later.

Formally, given a job j with deadline d_j, if the sampling duration takes 10% of a job's available deadline then $d_j^s = 10\% \times d_j$.

4.2 Assign Sampling Tasks to Instances

After a duration of the sampling phase is decided, tasks of an unknown application are assigned to one instance of each type. Ideally, it is possible to schedule the sampling phase on all existing VMs, hence no new ones are required, thus no additional cost. On the other hand, if there is any instance type with no available instance to receive sampling tasks, a new VM of this type must be created.

4.2.1 Calculate the Permissible Delay of an Instance

Before assigning sampling tasks to an existing instance, it is necessary to calculate how much workload an instance can receive. It should be noted that as a sampling should start as soon as possible, an existing instance must execute the sampling tasks immediately. In other words, an existing instance has to stop and delay an execution of its remaining tasks, i.e. execution preemption. However, delaying an execution of existing tasks may result in deadline violation. As a result, it is necessary to calculate the *permissible delay* of an instance, i.e. the amount of time to delay the execution of all existing tasks in an instance without resulting in deadline violation.

We assume that there is only one job of an unknown application is in execution of a time. In other words, when it is submitted, all other jobs are of known applications, i.e. their task execution times are known. Hence, it is possible to estimate the start and finish times of all tasks assigned to an instance.

For example, given an instance i that has to execute two tasks t_{j_1} and t_{j_2} of jobs j_1 and j_2 respectively. Moreover, the task execution times e_{j_1,it_i} and e_{j_2,it_i} are known. As a result, if i starts executing t_{j_1} at the time Γ_0, an execution can be estimated to finish at $\Gamma_0 + e_{j_1,it_i}$, which is also the time when i starts executing the next task t_{j_2}. Finally, the second task is completely executed at $\Gamma_0 + e_{j_1,it_i} + e_{j_2,it_i}$.

Given a task t which is assigned to an instance i, let $fi_{t,i}$ denote its estimated finish time. Moreover, let d_t be the deadline of a task, i.e. the deadline of a job containing a task. Hence, $d_t - fi_{t,i}$ is the amount of time from when a task's execution finishes until its deadline. In other words, it is also the permissible delay of a task's execution on an instance, i.e. a task's execution can be delay for $d_t - fi_{t,i}$ without resulting in deadline violation.

Given an instance i, let T_i be the list of its tasks which have not been executed yet, including a task which is currently in execution but not yet finished. In order to start executing the sampling task immediately on an instance, its remaining tasks must be delayed without resulting in any deadline violation. Hence, the permissible delay of all tasks on an instance is the minimum permissible delay of all tasks, which is calculated as: $pd_i = \min_{t \in T_i} (d_t - fi_{t,i})$.

4.2.2 Select and Assign Sampling Tasks to Instances

The process that selects and assigns sampling tasks to instances is described by Algorithm 2. Its inputs are all available instance types IT, all existing instances I, a submitted job j whose application is unknown, and the sampling duration d_j^s. Its objective is to assign sampling tasks to one instance of each type.

Algorithm 2 Assign Sampling Tasks

```
1:  function ASSIGN_SAMPLING(IT, I, j, d_j^s)
2:      for it ∈ IT do
3:          pd_it ← 0
4:          i_it ← null
5:          for i ∈ I do
6:              if it_i = it then
7:                  if pd_i ≥ d_j^s ∧ pd_i > pd_it then
8:                      pd_it ← pd_i
9:                      i_it ← i
10:                 end if
11:             end if
12:         end for
13:     end for
14:     for it ∈ IT do
15:         if i_it = null then
16:             i_it ← new instance of type it
17:         end if
18:         Assign sampling tasks to i_it
19:     end for
20: end function
```

First of all, Algorithm 2 tries to find existing instances which are able to receive sampling tasks (From Line 2 to Line 13). Notably, for each instance type, only one VM is selected. The selected instance is the one with highest permissible delay, which must also be greater than or equal to duration of the sampling phase, among other instances of the same type (Line 7).

It is possible to have instance types that have no available VM to receive sampling tasks. Those type may have no existing instances or their instances do not have sufficient permissible delay to receive sampling tasks. In either case, ew instances of those types are created (Line 16).

The number of sampling tasks assigned to each instance should be large enough to provide sufficient average task execution time, around 10 tasks should be enough. Notably, it is not necessary for all sampling tasks to be executed, as the rest of them will be executed in the full execution phase. Hence, assigning too many sampling tasks to an instance does not have any negative impact on overall performance.

4.3 Monitoring and Estimating the Task Execution Times

After assigning and start executing sampling tasks on instances, the execution is monitored in order to retrieve the actual execution of each task on an instance.

The sampling phase is completed when either all sampling tasks are executed or the sampling phase times out. After that, all the actual execution times of each tasks are used to estimate the task execution time of a job's application on all instance types.

After that, the task execution time of a job's application on an instance type is calculated as the average value of the actual execution times of all sampling tasks assigned to an instance of that type. However, if an instance is not able to execute even a single task, a very large value is used as the task execution time so that an instance type will not be

considered for scheduling.

With the estimated task execution times, a job with its remaining tasks, excluded executed sampling tasks, is scheduled to be executed within its deadline using the approached introduced in Section 3.

5. DYNAMIC REASSIGNMENT

The task execution time is used to estimate how long it takes an instance to execute one task. However, this value is not absolute but just an average estimation. In other words, two tasks of the same job may take the different amount of time to be executed on the same VM, which may result in unexpected delay, or even deadline violation. As a result, a dynamic reassignment mechanism is introduced.

Algorithm 3 Dynamic Assignment

```
1: function DYNAMIC_REASSIGNMENT(I)
2:     for i ∈ I do
3:         Update the estimated finish times of remaining
           tasks of i
4:         if ∃t ∈ T_i s.t. fi_t > d_t then
5:             if ∃i' ∈ I − {i} s.t. pd_i' ≥ e_t,i' then
6:                 Reassign t from i to i'
7:             end if
8:         end if
9:     end for
10: end function
```

Algorithm 3 presents the dynamic reassignment mechanism, which is performed periodically. First, the finish times of all tasks of each instance are updated (Line 3). A potential violation on an instance is detected if one of its tasks is estimated to finish after a deadline (Line 4).

A violated task is then moved to another instance whose permissible delay is greater than or equal the execution time of a task on that instance (Lines 5 and 6). In other words, the receiving instance is able to execute an additional task without resulting in a deadline violation.

Notably, it is possible that there is no available instance to receive tasks from a potentially violating instance. In that situation, the violation is unavoidable. This scenario will be investigated in the future work.

6. EVALUATION

In this section, we consider the implementation, the experimental set up and the results obtained.

6.1 Implementation

We have developed a Scala framework which is able to receive submitted jobs, perform scheduling and monitor the execution. The Model 6 is modelled and solved using Gurobi [1], an optimisation solver.

6.2 Experiment Setup

The setup of a trace-based simulation experiment is considered. In order to shorten the length of each experiment run, we define time blocks of 10 minutes (600 seconds). The instance creation overhead, i.e. Θ, is set to 15 seconds.

[1]http://www.gurobi.com/

Table 1: AWS Instance Types

Name	vCPU	ECU	Mem (GiB)	Storage (GB)	Price per Hour
m3.medium	1	3	3.75	4	$0.073
m3.large	2	6.5	7.5	32	$0.146
m3.xlarge	4	13	15	80	$0.293

6.2.1 Cloud Platform and Applications

The experiment is performed using three instance types provided by Amazon Web Service (AWS) cloud [2]. The hardware configurations and prices are presented in Table 1.

We use the execution trace of three real life applications and their task execution times on the selected AWS instance types are presented in Figure 1. The first application, denoted as app_1, is a file compression application using $lbzip2$[3] to compress data files ranging from $500MB$ to $1GB$. This application is I/O, memory and CPU intensive and supports parallelism. The performance is observed to improve when the application executes on multiple CPU cores although there is communication between cores.

The second application, denoted as app_2, is a machine learning application that executes sequentially and uses Support Vector Machines (SVM) for classification of a given dataset. This is facilitated by using a scientific package SVM^{light}[4]. The data sets are made available as input files ranging from $100MB$ to $500MB$. The application comprises I/O activity for reading a data file and CPU activity for training and classification. This is a sequential application and does not utilise multiple cores.

The last application, denoted app_3, is a Molecular Dynamics Simulation (MDS) of a 250 particle system in which the trajectory of the particles and the forces they exert are solved using a system of differential equations [6]. It is CPU intensive and embarrassingly parallel. Thus, its performance increases linearly with the number of cores in a VM.

Finally, the task execution times of the first two application have high variation since they perform operation on files of different sizes. On the other hand, MDS' task execution time has negligible variation as its operation is mostly CPU-bound.

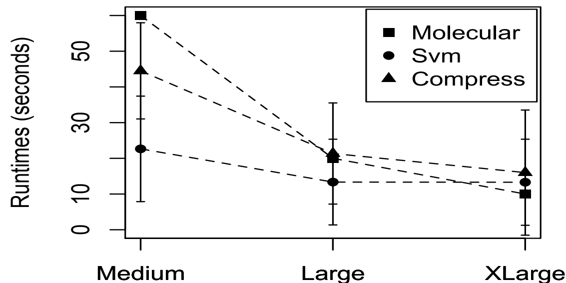


Figure 1: Task Execution Times

6.2.2 Evaluating Approaches

[2]http://aws.amazon.com/
[3]http://lbzip2.org/
[4]http://svmlight.joachims.org/

Table 2: Job Specification

Job	Application	Submission Time	Number of Tasks	Deadline
j_1	app_1	0	50	600
j_2	app_2	300	50	900
j_3	app_3	600	50	1200
j_4	app_1	900	100	1500
j_5	app_2	1200	100	1800
j_6	app_3	1500	100	2100

In order to evaluate our proposed approach, which schedules unknown BOT jobs with deadlines on the Cloud and is denoted as *unknown*, we also perform experiment using other settings. The first one is the ideal setting which has full knowledge regarding executing times of applications on all instance types and is denoted as *known*. In other words, this setting can directly use the mechanism presented in Section 3 to schedule the submitted jobs while the *unknown* one must perform the sampling phase described in Section 4 first.

Our proposed approach is also compared to an approach which uses fixed amount of resources and applies the round-robin method to distribute tasks. Instead of scheduling based on the task execution time, this approach assigns tasks to any idle VM. There are six different configurations: 8 instances of m3.medium (*medium*.8), 10 instances of m3.medium (*medium*.10), 4 instance of m3.large (*large*.4), 5 instances of m3.large (*large*.5), 2 instances of m3.xlarge (*xlarge*.2), and 3 instances of m3.xlarge (*xlarge*.3). These options are selected since the VMs cost per hour are quite similar.

6.2.3 Job Submission Pattern

Table 2 presents the job submission pattern used in our experiment. Overall, there are six jobs with two jobs for each application. Each job is submitted 300 second apart from each other. The first three jobs have 50 tasks each while the last three have 100 tasks each. All jobs have the same deadline, which is 600 seconds from their submission.

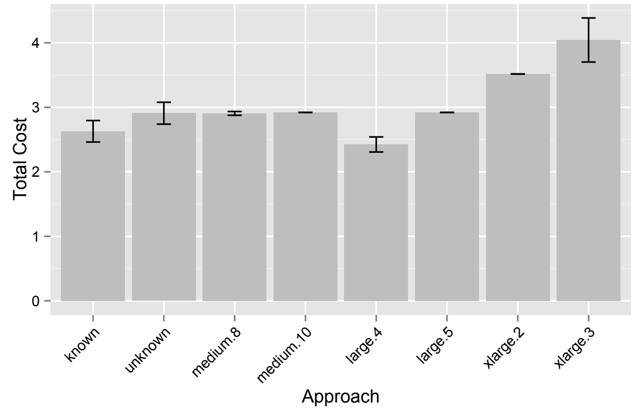
6.3 Experiment Results and Discussion

For each setting, the experiment is performed five times. Table 3 and Figure 2 presents the total monetary cost and the number of late tasks, i.e. finished after the job deadlines.

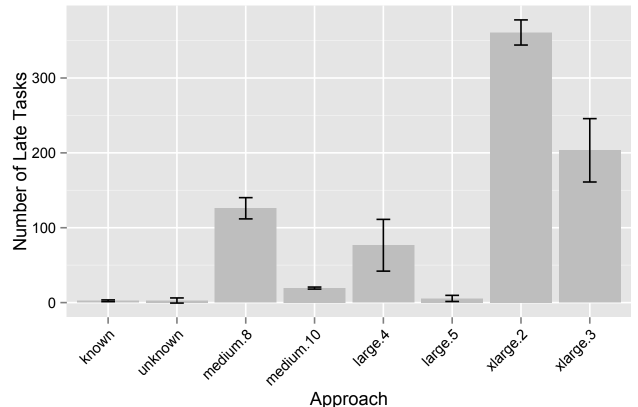
Table 3: Experiment Results

Approach	Mean Cost	Mean Number of Late Tasks
known	2.62	2.4
unknown	2.9	2.8
medium.8	2.91	126.0
medium.10	2.92	19.4
large.4	2.42	76.6
large.5	2.92	5.6
xlarge.2	3.51	360.8
xlarge.3	4.04	203.4

In comparison to the ideal *known* setting, the monetary cost of the *unknown* approach is 10% higher. Furthermore,



(a) Total costs



(b) Number of late tasks

Figure 2: Experiment result

its number of late tasks is 16% higher. On the other hand, the *unknown* approach out-performs other settings which use the fixed amount of resources.

The *medium*.8, *medium*.10 and *large*.5 settings have almost identical cost as the *unknown* one; less than 1$ more expensive. However, compared to the proposed approach, their numbers of late tasks are significantly higher, ranging from 2 (*large*.5) to 45 (*medium*.8) times higher. For nearly similar cost, our approach delivers superior quality of service when compared against other approaches.

Interestingly, using 4 instances of type m3.large results in the lowest cost, even lower than the *known* setting, which has full knowledge regarding task execution times. However, this setting also has a high degree of violation. On an average, the number of late tasks is 27 times higher than the proposed *unknown* approaches.

Finally, using either 2 or 3 m3.xlarge instances performs poorly in comparison with other approaches. They not only are the most expensive options but also have the highest numbers of late tasks. This can be explained using Figure 1 which presents the task execution times of all applications on all instance types. It can be seen that in comparison to a m3.medium (or m3.large) instance, a m3.xlarge one does not have any significant speed-up. Moreover, a SVM^{light} application has nearly identical performance on both of them.

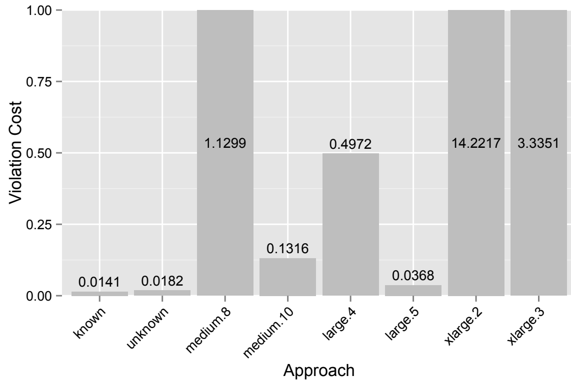


Figure 3: Violation Costs

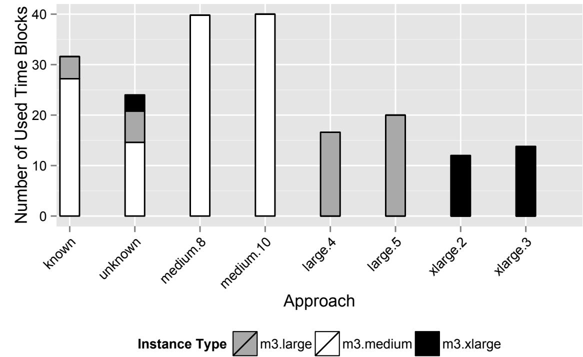


Figure 4: Used Time Blocks per Instance Type

On the other hand, because m3.xlarge is two and four times more expensive compared to m3.large and m3.medium, respectively, with the same amount of money, the number of m3.xlarge instances is always less than the number of instances of other types, which results in lower execution parallelism, since each instance can only execute one task at a time. In summary, using instances of m3.xlarge has insignificant performance speed-up per instance but significantly reduces the number of instances to execute tasks in parallel. As a result, its overall performance is lower in comparison to other settings.

In order to provide a better comparison between all approaches, we introduce the *violation cost* metric which represents the violation of a execution in term of monetary cost. First of all, the cost of each non-violated tasks can be calculated by dividing the number of non-violated tasks to the total cost:

$$cost_per_tasks = \frac{total_cost}{number_of_non_violating_tasks} \quad (7)$$

Then, the violation cost is calculated as:

$$violation_cost = cost_per_tasks \times number_of_late_tasks \quad (8)$$

The violation cost of an approach can also be described as the additional cost required to finish all tasks within their deadline. Hence, it is desirable to achieve as low violation cost as possible.

As shown by Figure 3, the violation costs of the *unknown* approach is 29% higher than the *known* one. This is understandable since our approach not only costs more but also has the higher number of late tasks. On the other hand, the proposed approach still outperforms the rest, whose violation costs are two (e.g. $large.5$) to more than 700 times higher (e.g. $xlarge.3$).

In order to understand why the *unknown*, and *known*, approaches outperform other ones which use the fix amount of VMs of the same type, Figure 4 illustrates the overall number of used time blocks corresponding to each instance types for each approaches.

It can be seen that, apart from the settings which use the same instance types, the *known* and *unknown* approaches use the combination of different ones. In other words, using the model described in Section 3, those two approaches are able to select the suitable amount of resources to execute given workload while ensuring the quality of service, i.e. deadline.

Notably, the *known* approach does not use any instance of type m3.xlarge which is expensive without significant speed-up, as explained earlier. On the other hand, since the sampling phase requires tasks to be executed on instances of all types, the *unknown* approach has to create some instances of m3.xlarge. Due to the cost-inefficiency of this type which results in lower execution parallelism without significant performance improvement, our proposed approach is outperformed by the ideal one.

Finally, we evaluate the resource mis-utilisation of each approach. There are two types of mis-utilisation: *Resource over-utilisation* is when the amount of allocated resources is more than necessary and results in idle VMs. The over-utilisation is calculated as the total idle times of all instances. Notably, as showed in Section 6.2.3, jobs are submitted in the way so that their execution overlap with each other. Hence, ideally, there should not be any idle instances.

On the other hand, *resource under-utilisation* happens when the amount of resources is not sufficient to execute all tasks within their deadlines. As a result, resource under-utilisation is calculated as the total violation time.

Figure 5 illustrates the average resource mis-utilisation of each approach. It can be seen that both the *known* and *unknown* manage to achieve very low resource over- and under-utilisation in comparison with other ones.

On the other hand, most of the fixed resources setting have high under-utilisation due to the fact that they have enough resources to finish some jobs very early before the submission of the next one. However, occasionally, they do not have enough resources to execute some other jobs within deadlines, hence the over-utilisation is also very high.

Finally, the $xlarge.2$ setting has no idle time but very high late time. In other words, the allocated resources of this setting is always less than required.

In summary, the *known* and *unknown* approaches are able to not only achieve low cost but also minimise violation by flexibly adjust the amount of allocated resources based on current workload.

7. CONCLUSION

In this paper, we investigated the problem of scheduling multiple unknown BoT jobs with deadlines on the cloud so

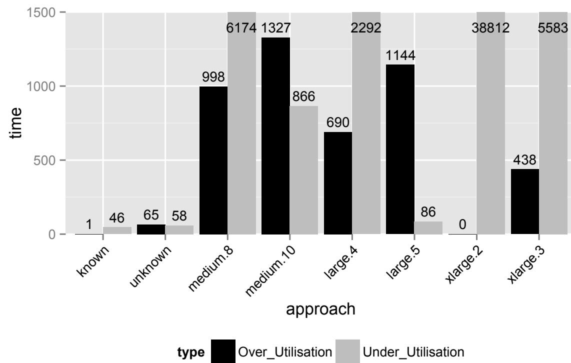


Figure 5: Resource mis-utilisation

that the total monetary cost can be minimised. Our proposed approach retrieves required knowledge through a sampling phase and is able to deliver a scheduling decision within 10% cost and 16% violation compared to the ideal setting which has prior knowlesge. It also outperforms other approaches that use a fixed amount of resources by reducing the monetary cost of violation by at least two times.

In the future, we are planning to develop an approach to support scheduling multiple jobs, known or unknown, at once instead of one at a time. It will be interesting to investigate other methods to find task execution times of an unknown job besides performing sampling.

8. REFERENCES

[1] Amazon web service.

[2] A. Amato, B. Di Martino, and S. Venticinque. Cloud brokering as a service. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 9–16, Oct 2013.

[3] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[4] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *Computers, IEEE Transactions on*, 59(2):202–217, Feb 2010.

[5] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, Oct. 2014. USENIX Association.

[6] K. Bowers, E. Chow, H. Xu, R. Dror, M. Eastwood, B. Gregersen, J. Klepeis, I. Kolossvary, M. Moraes, F. Sacerdoti, J. Salmon, Y. Shan, and D. Shaw. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 43–43, Nov 2006.

[7] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[8] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.

[9] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 459–471, Santa Clara, CA, July 2015. USENIX Association.

[10] J. O. Gutierrez-Garcia and K. M. Sim. Agent-based cloud bag-of-tasks execution. *Journal of Systems and Software*, 104:17 – 31, 2015.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[12] A. Iosup and D. Epema. Grid computing workloads. *Internet Computing, IEEE*, 15(2):19–26, March 2011.

[13] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430, June 2012.

[14] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48, Oct 2010.

[15] A. Oprescu and T. Kielmann. Bag-of-tasks scheduling under budget constraints. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 351–359, Nov 2010.

[16] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[17] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 352–358, 2002.

[18] L. Thai, B. Varghese, and A. Barker. Budget constrained execution of multiple bag-of-tasks applications on the cloud. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 975–980, June 2015.

[19] L. Thai, B. Varghese, and A. Barker. Task scheduling on the cloud with hard constraints. In *Services (SERVICES), 2015 IEEE World Congress on*, pages 95–102, June 2015.

[20] J. Ward and A. Barker. Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1), 2014.