A Marketplace for Cloud Resources

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Anmol V. Singh
IST Austria
anmol.tomar@ist.ac.at

Vasu Singh
IST Austria
vasu.singh@ist.ac.at

Thomas Wies
IST Austria
thomas.wies@ist.ac.at

Damien Zufferey
IST Austria
damien.zufferey@ist.ac.at

ABSTRACT

Cloud computing is an emerging paradigm aimed to offer users pay-per-use computing resources, while leaving the burden of managing the computing infrastructure to the cloud provider. We present a new programming and pricing model that gives the cloud user the flexibility of trading execution speed and price on a per-job basis. We discuss the scheduling and resource management challenges for the cloud provider that arise in the implementation of this model. We argue that techniques from real-time and embedded software can be useful in this context.

Categories and Subject Descriptors

H.1 [**Information Systems**]: Models and Principles; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Design, Performance

Keywords

Cloud computing, IaaS, pricing models, large-scale scheduling, worst-case execution time

1. INTRODUCTION

Computing services that are provided by datacenters over the internet are now commonly referred to as *cloud computing*. Cloud computing covers different services: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). For example, IaaS is provided by the Amazon Elastic Compute Cloud (EC2) [1], PaaS is provided by the Google App Engine [6], and SaaS is provided by Force.com [14]. Today, while pricing models for PaaS and SaaS are oriented towards benefiting the customer, the pricing models for IaaS are oriented towards covering the costs incurred by the service provider. A good example of this is the Amazon EC2 pricing model. Here a user, instead of paying for actual CPU-hours, has to pay for the hours she rents the machines, no matter whether the machines are idle or busy. An IaaS provider wants to gain revenue for the time its machines are rented, and not just for the time they are being used. Both Amazon EC2 [1] and Microsoft Windows Azure [17] charge fixed prices for computing power, storage, and data transfer. Recently, there has been an effort to develop a programming and pricing model, called FlexPRICE [7], which brings together a long-term pricing contract preferable to the IaaS provider, and a job-oriented pricing model preferable to the IaaS user.

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.

1.1 FlexPRICE

We illustrate the working of FlexPRICE using an example. Imagine that a user wants to use ImageMagick [8] to apply an image transformation on a set of images in a data store. The transformation is composed of the ImageMagick transforms `paint`, `emboss`, and `average`. To every image she first applies the `paint` and `emboss` transforms separately, producing two new intermediate images. Then she uses the `average` transform to average the intermediate images together with the original image into a single new output image. The final image is stored back into the data store. This is a MapReduce job [4]. All the transformations are mappers. The reducer is an identity function.

A user writes a job. Figure 1 shows the description of this job in a language that enables the user to describe jobs as data flow graphs. A job consists of *schemas* that describe templates of tasks, and *connections* that describe the primary inputs and outputs of the job, and how the tasks interact. In a task description the user specifies estimates for the task's resource requirements, such as execution time and memory consumption. These requirements can be specified as simple functions in terms of the size of input data objects. For instance, in our example the user specifies that each `paint` task requires around 20 seconds per MB of the size of the input image. As we shall discuss later, worst-case execution time (WCET) analysis [13, 16] may be more reliable than user provided estimates for execution time.

The cloud computes a price curve. Once a user submits a job to the cloud, the cloud uses the data store queries specifying the primary inputs of the job to expand the job description into an *execution plan*. The execution plan is a directed acyclic graph of tasks (instantiated schemas) and data objects (instantiated connections). Figure 2 shows part of an execution plan for the job in Figure 1: each of the three task schemas results in one task per image that is stored in the data bucket `img_buc`.

```
// schemas
mapper pnt ([i1]) ([o1]) {
  duration 20 * i1
  memory 200
  o1 = i1
  binary 'convert -paint 10'
}
mapper emb ([i1]) ([o1]) {
  duration 10 * i1
  memory 200
  o1 = i1
  binary 'convert -emboss 10'
}
mapper avg ([i1], [i2], [i3]) ([o1]) {
  duration 3 * (i1 + i2 + i3)
  memory 200
  o1 = i1
  binary 'convert -average'
}
// connections
pnt.o1 = avg.i1
emb.o1 = avg.i2
pnt.i1 = match * from img_buc
emb.i1 = match * from img_buc
avg.i3 = match * from img_buc
avg.o1 = store $avg.i3 into res_buc
```

Figure 1: Job description for a composed image transformation that is applied to a set of images in a data store

Using the specified resource estimates, the cloud provider computes a selection of possible schedules for executing the execution plan on the cloud. Note that while some schedules may parallelize the mapper (transformation) tasks, some schedules may execute the mappers sequentially. Different schedules have different finish times. The cloud provider has an associated pricing model to quote prices of the schedules. The pricing model indicates the price of computation per unit time and the setup price per machine, and the price of data transfer per unit size. An important aspect of the pricing model is a time discount factor, which lets the cloud provider offer a discount for delayed execution. This allows users to get their jobs executed for a cheaper price if they are willing to wait, and provides an opportunity for the cloud to dynamically schedule the user job in dull periods of resource utilization. After computing the different schedules and their prices, the cloud provider presents these schedules to the user in the form of a price curve as shown in Figure 3. Intuitively, the price curve represents the tradeoff between the finish time and price of execution of the job. Moreover, the price curve hides from the user, the execution details like the number of machines used, and the time of execution of individual tasks.

Tasks

Task id	Duration	Memory
1	180 s	200 MB
2	60 s	200 MB
3	54 s	200 MB
...	...	...

Objects

Object id	Source	Destination	Size
1	img_buc/1.jpg	1.i1	6 MB
2	img_buc/1.jpg	2.i1	6 MB
3	img_buc/1.jpg	3.i1	6 MB
4	1.o1	3.i2	6 MB
5	2.o1	3.i3	6 MB
6	3.o1	res_buc/1.jpg	6 MB
...	...	...	...

Figure 2: An execution plan for the user job in Figure 1

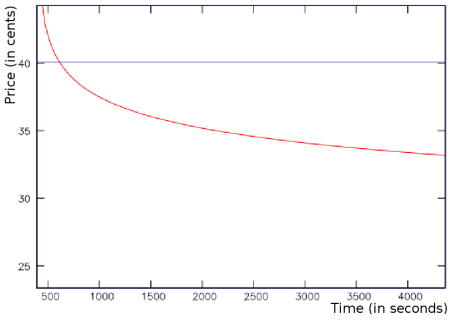


Figure 3: A price curve as presented by FlexPRICE to the user

The user chooses an execution. The user may choose any point on the price curve. Let the point have a deadline d and a price p. The chosen point results in the following contract between the user and the cloud provider: If every task of the job meets the user-specified duration, the cloud provider must finish the job before time d, and not charge the user more than p. Note that at this point, we do not formulate the consequences of a task exceeding its specified duration.

1.2 Benefits of FlexPRICE

We now highlight the benefits of FlexPRICE. It offers an economic advantage and ease of use for the user. Moreover, FlexPRICE helps to improve the predictability of the resource usage for the cloud provider.

Economic benefits. The core idea of FlexPRICE is to allow the cloud provider to discount delayed computation, and let users choose the price and deadline of their jobs. While the delay brings greater predictability of resource utilization for the provider (as described below), the discount benefits the user. If a user is willing to wait, she can execute the job at a lower price with FlexPRICE.

Secondly, current cloud providers like Amazon EC2 [1] charge users with a per-hour billing granularity. This generates an inequality in the average price of jobs per unit time. Smaller jobs that are short of a full hour pay a higher average price. This discourages the user to exploit parallelism when the tasks of the job are small. Ideally, the user would like to pay for the actual amount of computation done. Except for the overheads of parallelization, the price of a parallel schedule should not differ from that of a sequential schedule. The finer per-second billing granularity used by FlexPRICE benefits the user.

Simpler view of the cloud. In addition to the economic advantage, FlexPRICE offers users a simple programming interface to the underlying computing resources. Such an interface is appealing to the user who now only sees what is important to her (e.g. deadline, price), and does not see

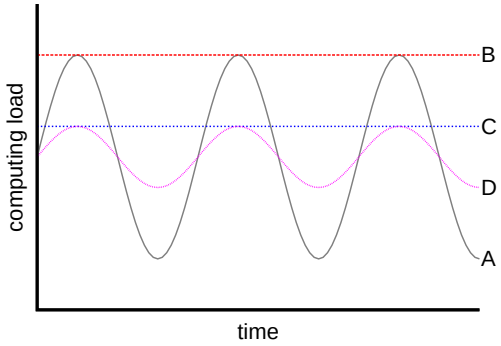


Figure 4: FlexPRICE offers greater predictability of resource utilization for the cloud provider

what she does not care about (e.g., the precise schedule of the job). Thus, FlexPRICE offers a simple abstraction of the cloud infrastructure to the user.

Greater predictability. Cloud providers prefer to have a knowledge of the load profile, that is, the distribution of input jobs in time. This is evident from the renting policies of Amazon EC2. For example, to rent more than twenty instances, one has to seek permission from Amazon [1]. Similarly, renting a large number of instances requires a long-term contract. In practical scenarios, a cloud provider may not be able to predict the number of input jobs in the future. The input job sequence may exhibit a high variance, depending upon the user behavior: there might be peak durations when many users simultaneously submit jobs and dull periods when there are no input jobs (see curve A in Figure 4). The flexibility that FlexPRICE offers to IaaS users results in better predictability of the cloud resource usage. FlexPRICE allows users to choose different execution deadlines for their jobs. If some users agree to delay the computation for a lower price, FlexPRICE can schedule the computation in the dull hours, in effect, balancing the overall load on the system. This allows FlexPRICE to reduce the variance in the computing requirements as depicted by curve D in Figure 4. Note that to serve the users according to the input load, the cloud provider has to provision resources as depicted by line B. However, if users choose execution with FlexPRICE, the resource provisioning requirement drops to line C.

Implementing a platform like FlexPRICE requires to solve many challenging problems. This paper formalizes the problem and discusses the challenges involved in the implementation. We discuss how many of these challenges relate to research problems in real-time and embedded systems. Furthermore, we present a basic prototype to solve the scheduling problem that arises in FlexPRICE.

2. FRAMEWORK

We start with a formal description of the problem. We first describe a cloud infrastructure. Then, we formalize execution plans and schedules of execution plans on the cloud infrastructure.

2.1 Cloud

A cloud is a term used broadly for the infrastructure of a datacenter – cpus, network, and other peripherals. In our model, we represent a cloud as a fully connected graph of networked computation and data nodes. We assume that there exists a communication link between each pair of nodes. We also assume that each link has an individual bandwidth and the data transfer on one link does not affect the other links. This assumption allows us to separate the orthogonal issue of distribution of total bandwidth across the links from our work.

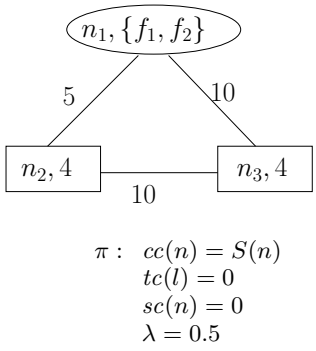


Figure 5: An example cloud

Let F represent the set of data files in the cloud. These files are primary inputs and outputs for the user jobs. Formally, we define a *cloud* as $C = \langle N, L, S, B, data, \pi \rangle$ where

- $N = N_d \cup N_c$ is the set of nodes, where N_d is the set of data store nodes, and N_c is the set of compute nodes,
- $L = N \times N$ is the set of communication links between the nodes,
- the function $S : N_c \rightarrow \mathbb{N}$ represents the speed of the compute nodes in the cloud,
- the function $B : L \rightarrow \mathbb{N}$ represents the bandwidth of every link in the cloud,
- the function $data : N_d \rightarrow 2^F$ represents the set of files stored on every data node, and
- the pricing model π determines how users are charged for using the cloud. The pricing model is formally defined later.

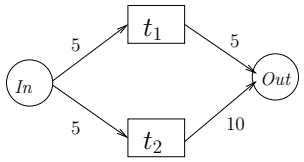
Example. Figure 5 shows an example of a cloud. The cloud is depicted by the graph in the upper part of the figure. The numbers on the links represent their respective bandwidths. The compute nodes, shown in rectangles, contain an identifier and their respective speed. The data node, shown in an ellipse, contains an identifier and its data files.

2.2 Execution plan

Users submit jobs to be executed on the cloud. Using the data store, the job is unfolded to obtain an execution plan as described in the example in Section 1.1. An execution plan is a graph consisting of independent tasks as nodes and data transfers between them as edges. A task corresponds to a piece of computation in a user program. An object is a piece of data that is transfered from one task to another.

Formally, an *execution plan* is a directed acyclic graph (DAG) $E = \langle T, O, D, Z, file \rangle$ where

- $T = T_d \cup T_c$ is the set of tasks, where T_d is a set of data tasks and T_c is a set of computation tasks,



T	D	*file*
In	-	f_1
t_1	12	-
t_2	24	-
Out	-	f_2

Figure 6: An execution plan

- $O \subseteq T \times T$ is the set of objects between the tasks,
- $D : T_c \to \mathbb{N}$ gives the duration of the computation tasks on a unit speed node in time units,
- $Z : O \to \mathbb{N}$ gives the size of the objects in terms of the unit data size, and
- $file : T_d \to F$ specifies the files associated with the data tasks.

Intuitively, the data tasks represent the reading of the primary inputs and the writing of the final outputs.
Example. Figure 6 shows an example of an execution plan. The numbers on the edges represent the size of the objects. The compute tasks are shown in rectangles and the data tasks in circles. The adjacent table lists the duration of the computation tasks and the files associated with the data tasks.

2.3 Schedule

A schedule maps the set of tasks in an execution plan to nodes and starting time of execution on the nodes. Formally, given an execution plan E and a cloud C, a *schedule* $\sigma : T \to N \times \mathbb{Q}^+$ assigns tasks of E to the nodes in C, at specific start times.

In general, an execution plan can be executed on a cloud with many different schedules. Let Σ be the set of all schedules of the execution plan E on cloud C. We define a *scheduling algorithm* by the function $A : C \times E \to 2^{\Sigma}$. Intuitively, a scheduling algorithm takes a cloud and an execution plan, and returns a set of possible schedules.

Given an execution plan $E = \langle T, O, D, Z, file \rangle$, a cloud $C = \langle N, L, S, B, data, \pi \rangle$, and a schedule σ for E on C, we say that the schedule σ is *well-formed* if all of the following hold:

- for every compute task $t_c \in T_c$, we have $n_c \in N_c$, where $\sigma(t_c) = (n_c, s)$ for some $s \in \mathbb{Q}^+$
- for every compute node $n \in N_c$, there do not exist tasks t and t' such that $\sigma(t) = (n, s)$ and $\sigma(t') = (n, s')$ and $s' < s + D(t)/S(n)$
- for every data task $t_d \in T_d$, we have $n_d \in N_d$ and $file(t_d) \in data(n_d)$ where $\sigma(t_d) = (n_d, s)$ for some $s \in \mathbb{Q}^+$
- for every object $(t_1, t_2) \in O$ where $t_1 \in T_c$, we have

$$s_2 > s_1 + \frac{D(t_1)}{S(n_1)} + \frac{Z((t_1,t_2))}{B((n_1,n_2))}$$

  where $(n_1, s_1) = \sigma(t_1)$ and $(n_2, s_2) = \sigma(t_2)$
- for every object $(t_1, t_2) \in O$ where $t_1 \in T_d$, we have

$$s_2 > s_1 + \frac{Z((t_1,t_2))}{B((n_1,n_2))}$$

  where $(n_1, s_1) = \sigma(t_1)$ and $(n_2, s_2) = \sigma(t_2)$

Informally, every compute task is scheduled on a compute node and is isolated from other compute tasks on the node, every data task is scheduled at a data node where the task's data file exists, and every task is scheduled to start at a time when all predecessor tasks are finished and there is sufficient delay for their outputs to reach the task.

2.4 Pricing model

A *pricing model* π of a cloud C determines how a user is charged for executing a schedule on the cloud. Formally, π is a tuple $\langle cc, tc, sc, \lambda \rangle$ where

- the $cc : N_c \to \mathbb{Q}$ determines the *computation cost* for unit time execution on a given node,
- the function $tc : L \to \mathbb{Q}$ computes the *data transfer cost* for transferring a unit size object over a given communication link,
- the function $sc : N \to \mathbb{Q}$ determines the *setup cost* for each node, and
- $\lambda \in \mathbb{Q}$ denotes the *time discount factor* for the cloud.

Example. The lower part of Figure 5 shows the pricing model of the cloud. The computation costs of a task on a node are linear in the speed of the node. There are no transfer and no setup costs. The time discount factor is 0.5.

Given a pricing model π, we define a function *price* $P_\pi : \Sigma \to \mathbb{Q}$ that gives the price incurred by the cloud for executing a schedule for a given execution plan. The price of a schedule is given by the sum of its total computation costs, total data transfer costs, and total setup costs, scaled by the time discount factor. Formally, the function P_π is defined as follows:

$$P_\pi(\sigma) = e^{-\lambda \cdot t_0} \cdot P'_\pi(\sigma)$$

where

$$t_0 = \min_{t \in T} \sigma(t)_{\#2}, \text{ and}$$

$$P'_\pi(\sigma) = \sum_{t \in T} cc(\sigma(t)_{\#1}) \cdot D(t) + \sum_{(t_1,t_2) \in O} tc(\sigma(t_1)_{\#1}, \sigma(t_2)_{\#1}) \cdot Z((t_1,t_2)) + \sum_{n \in un(\sigma)} sc(n)$$

Here $x_{\#i}$ denotes projection of a tuple x to its i-th component. The set of *used nodes* $un(\sigma)$ of a schedule σ is defined by $un(\sigma) = \{\sigma(t)_{\#1} \mid t \in T\}$.

3. CHALLENGES

Solving the scheduling problem in this setting requires to address many issues. First of all, the scheduling requires an estimate of the duration for all tasks. Given the duration, efficient static scheduling techniques need to be developed that scale with the size of the cloud. As tasks may finish in a fraction of the assigned duration, the cloud needs to deploy dynamic scheduling in order to optimize the execution. We discuss these issues one by one.

3.1 Estimation of task duration

To schedule the tasks in the cloud, we require the users to provide an estimate of how long the tasks may last in the worst case. To successfully execute the whole execution plan, it is essential to finish every task within its assigned duration. Thus, it is important to allocate sufficient duration for every task. On the other hand, note that both the price and the finish time of the job depend on the individual durations of the tasks. Thus, it is important that the duration is not arbitrarily large as it affects the price curve presented to the user. So, a conservative bound on the worst case execution time is essential. This forms an interesting application area of the research in WCET analysis [13, 16].

3.2 Hardness of scheduling

In order to sample the space of possible schedules for an execution plan, we need to solve optimization problems subject to multiple objective functions. Many of the optimization problems in the domain of scheduling are NP-hard [12]. For example, given a cloud C and an execution plan E, computing the deadline-optimal or price-optimal schedule are NP-hard problems. Similarly, given a maximum deadline (resp. price), computing the cheapest (resp. fastest) schedule are shown to be NP-hard problems. Thus, instead of computing optimal schedules we use scheduling heuristics that produce good approximations of the optimal schedules.

The literature on static scheduling techniques is relevant in this context. Work on static multiprocessor scheduling dates back to 1977 [15], where the problem of scheduling a directed acyclic graph of tasks to two processors is solved using network-flow algorithms. Further research in this direction focused on scheduling distributed applications on a network of homogeneous processors [10]. A wide range of static scheduling heuristics have been classified and rigorously studied [9].

3.3 Size of the cloud

The scheduling heuristics usually scale linearly in the size of the cloud and the execution plan. However, due to the large size of the cloud, the computation of the schedules may pose large overheads. We therefore must decouple the complexity of the employed scheduling algorithms from the size of the cloud. We may use *abstract* representations of the cloud in order to reduce the scheduling overheads, as we discuss later in our prototype in Section 4.1. Techniques from model checking using abstraction refinement [3] can also be borrowed in this context. For example, given a cloud C, we may first build an abstraction C_a of the cloud. Given an execution plan E, we first find a schedule σ on the cloud C_a. Then, we check whether σ is well-formed on C. If not, we refine the abstraction C_a of the cloud, and recompute a new schedule for E.

3.4 Dynamic scheduling techniques

The statically computed schedules depend on the user's estimation or the worst-case execution analysis of the tasks. Thus, the static schedules only give rough upper bound guidelines for the execution. There are indeed many opportunities to further optimize the statically computed schedules at runtime. For example, when tasks finish long before their estimated durations, other tasks whose dependencies are already met may be used to fill the emerging idle time slots. Such dynamic rescheduling is essential for achieving good utilization of the resources. Techniques from run-time scheduling can be applied in this context. For example, backfilling is a common technique that pulls forward the execution of tasks to optimize the performance [11, 5].

We now present our prototype solution that addresses the challenges in scheduling of FlexPRICE. At this point, we do not look into the WCET analysis, and rely on user provided task durations for scheduling.

4. OUR PROTOTYPE

We build a prototype scheduler that relies on a combination of static and dynamic scheduling. Also, to reduce overhead, we use an abstract representation of the cloud.

4.1 Static scheduler

Scheduling heuristics. Based on the existing literature on scheduling heuristics, we implemented the following schedulers: (i) a *greedy scheduler* which takes two tuning parameters α and β and at every step chooses the node which minimizes the sum of α times the price of the assignment and β times the starting time of the assigned task. Tuning the parameters α and β gives a whole spectrum of greedy schedules from fast expensive schedules to slow cheap schedules, (ii) a *deadline division scheduler* [19] which takes a deadline as a parameter and applies a distribution of the deadline over task partitions, where a partition is a set of connected tasks in the job, (iii) a *clustering scheduler* based on the Dominant Sequence Clustering (DSC) algorithm [18] for scheduling task DAGs on multiprocessors. Different heuristics vary in the optimization parameters, and thus generate many different schedules.

Abstract cloud representation. The amount of time required to compute the schedules should be low. It should not increase as the number of nodes in the cloud increases. To achieve this, we abstract the information about the nodes and keep a succinct representation of a cloud. Generally, several nodes have a similar configuration in a cloud, and there are only a few configurations. To start with, we treat each configuration as an equivalence class, and thus represent a cloud by a set of these classes.

Instead of choosing a node to schedule a task, our heuristics first choose an equivalence class of nodes. In a second step, the scheduler returns the node in that equivalence class that can fetch the input objects from the predecessor tasks at the earliest. Note that as we sort the tasks in the topological order, the predecessors of a task are scheduled before a task is scheduled. We sort the nodes in every equivalence class in the order of earliest availability. Note that the abstraction of the cloud gives an overapproximation of the cloud, that may result in less optimal solutions than those obtained with the concrete cloud representation.

4.2 Dynamic scheduler

We use a combination of different dynamic scheduling techniques. The compute nodes make local scheduling decisions by dynamically reordering assigned tasks. Larger idle slots on individual nodes are reported back to the central static scheduler which then reschedules entire pending jobs to fill free slots.

Backfilling. The compute nodes play an important role in optimizing the scheduling. We rely on techniques of backfilling [11, 5] to dynamically optimize the task-level schedul-

Job	Execution plan		12 node cloud		80 node cloud	
	#tasks	#objects	Overhead	Schedules	Overhead	Schedules
Gene Sequencing	11	22	FM: 0.012	F: (1041, 881.66)	0.023	F: (449, 2209.37)
			PC: 0.413	S: (12437, 283.32)	0.701	S: (12434, 98.91)
	21	42	FM: 0.029	F: (1080, 1732.58)	0.039	F: (466, 4334.71)
			PC: 0.632	S: (12904, 684.90)	0.982	S : (12906, 194.42)
Machine Learning	183	550	FM: 0.223	F: (16, 60.43)	0.450	F: (6. 113.49)
			PC: 4.021	S: (212, 15.11)	7.970	S: (164, 6.34)
	6711	7732	FM: 8.811	F: (166, 856.46)	14.670	F: (20, 2018.98)
			PC: 256.866	S: (2310, 235.87)	346.988	S: (1498, 97.10)
Population Genetics	22	45	FM: 0.021	F: (1254, 5280.18)	0.042	F: (504, 13200.14)
			PC: 0.613	S: (17544, 1599.43)	0.961	S: (25044, 627.69)
	210	421	FM: 0.236	F: (10004, 44641.29)	0.495	F: (1254, 103216.87)
			PC: 4.112	S: (145044, 16156.28)	8.102	S: (122544, 6159.20)
Image Processing	401	802	FM: 0.328	F: (498, 2345.70)	0.820	F: (66, 5524.51)
			PC: 10.840	S: (7272, 1115.15)	24.896	S: (6244, 315.87)
	2005	2406	FM: 1.507	F: (1341, 6872.08)	2.908	F: (147, 15352.13)
			PC: 50.555	S: (20049, 2918.50)	94.715	S: (17377, 941.92)

Table 1: Evaluation of the static scheduler. We consider two execution plans for every class of jobs. We give the schedule computation overhead in seconds for both modes: Fastest scheduling mode (FM) and Price curve mode (PC). For each case, we list the fastest and the slowest schedule obtained in the price curve mode: a schedule is expressed as (D, P), where D is the finish time (in seconds) of the job, and P is the price (in cents) of the job.

ing. A task may finish earlier than its assigned duration, and this optimization enables possibly recollecting the remaining reserved time interval for the task, thus expanding the free interval on a compute node. Every compute node maintains a priority queue for all tasks that are scheduled on it, ordered by the task start times. When a task finishes earlier than its specified duration, the compute node checks whether the next task in the queue has to start immediately. This check is essential to avoid possible interference with other schedules on the node. When a task is chosen for backfilling, and in addition, there are already sufficient resources available, then it is immediately executed. Otherwise, the compute node informs the static scheduler that the completed task finished before the estimated time. This information is then used by the static scheduler to update the state of the symbolic cloud, and allow job-level rescheduling. The backfilling strategy ensures that the static scheduler is only informed about sufficiently large idle intervals between scheduled tasks.

Rescheduling of jobs. Intuitively, the user choice reflects the amount of freedom to the scheduler in execution. After a schedule is chosen by a user, the user is informed about the maximum duration and the maximum price the user has to pay. The schedule is sent to the compute nodes only if the schedule starts within a threshold, say a few minutes. If the schedule is not starting within the threshold, the static scheduler adds the job to a queue of jobs that are candidates for rescheduling. For example, due to backfilling, the scheduler might find new slots and start jobs scheduled in the future at an earlier time.

Load profiling. The rescheduler is a key to reducing load variability. In the same spirit as [2] we use some knowledge about the expected load of the system to maximize the number of users the cloud can serve. The scheduler maintains a profile of job input through time. The rescheduler enables dynamic scheduling which leads to reduced load variability. The load profiler triggers the rescheduling if the expected load goes below a certain threshold. The rescheduler checks for jobs scheduled during a peak duration, and pulls the job to the current time. As the rescheduler never pushes a job into the future, the guarantees of duration given to the user are met for all jobs.

4.3 Evaluation

We evaluated the static scheduler of our prototype using several jobs from different computing domains: gene sequencing, machine learning, population genetics, and image processing. All jobs are MapReduce jobs, however the number of tasks and objects in the execution plan varies for every job.

Table 1 plots the time required by our scheduler to obtain the schedules, and the details of the fastest and the slowest schedules obtained. We run the scheduler in two modes: fastest mode to get the fastest schedule, and price curve mode to get a range of schedules. We consider two cloud scenarios: one of 12 nodes with two different configurations, and another of 80 nodes with six different configurations. Note that computing the price curve is time-consuming for the largest jobs we consider. In future work, we plan to use abstract representation for the execution plan so that our scheduler can handle large jobs with low overheads.

5. CONCLUSION

FlexPRICE is a novel approach to scheduling jobs on the cloud by separating the concerns of the end users and the cloud provider and bridging the gap between the pricing needs of the two sides. The idea is to abstract away the unnecessary details from the user – namely, the scheduling complexity – and give her the illusion of using a large single computer. The features a user is interested in, like the price and the deadline of execution, are kept transparent to her that allows her to choose her preferred execution. Leaving the responsibility of scheduling the jobs with the cloud provider enables the provider to achieve good utilization of the cloud resources.

However, finding different schedules favorable both to the user and the cloud provider requires to solve many challenging problems, like estimating the job execution times and scheduling tasks at the massive scale of a cloud. We believe that several pieces of research in embedded and real-time systems can be applicable to these problems.

6. REFERENCES

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`.

[2] A. Bestavros. Load profiling: a methodology for scheduling real-time tasks in a distributed system. In *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 449, Washington, DC, USA, 1997. IEEE Computer Society.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, pages 107–113, 2008.

[5] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *IPPS/SPDP*, pages 542–546, 1998.

[6] Google App Engine. `http://code.google.com/appengine/`.

[7] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. FlexPRICE: Flexible provisioning of resources in a cloud environment. In *IEEE International Conference on Cloud Computing CLOUD*. IEEE, 2010.

[8] ImageMagick. `http://www.imagemagick.org`.

[9] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, pages 406–471, 1999.

[10] C.-H. Lee and K. G. Shin. Optimal task assignment in homogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 119–129, 1997.

[11] D. A. Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.

[12] C. Papadimitrious and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513, New York, NY, USA, 1988. ACM.

[13] P. P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[14] Enterprise cloud computing. `http://www.salesforce.com/platform`.

[15] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, pages 85–93, 1977.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[17] Windows Azure Platform. `http://www.microsoft.com/windowsazure`.

[18] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 951–967, 1994.

[19] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *International Conference on e-Science and Grid Computing*, pages 140–147. IEEE Computer Society, 2005.