THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A STUDY OF PRICE AND CAPACITY TRADE-OFFS OF REPLICATING COMPUTATION ON THE PUBLIC CLOUD

QIANLIN LIANG
SPRING 2016

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Area of Computer Science

Reviewed and approved* by the following:

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Jesse Barlow
Professor of Computer Science and Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Amazon EC2 spot market allows tenants to bid for various type of instances, which may be revoked if the spot prices rise too high. However, because of this risk, instances in spot market are significantly cheaper (by as much as 10 times) than their corresponding on-demand instances. To mitigate this risk, a lot of spot price modeling (e.g. auto-regressive model, Markov-chain model) and fault-tolerance methods (e.g. checkpointing method, replicating method) are proposed in prior work.

In this thesis, we design a batch job computation service for spot market using replicating fault-tolerance technique. We also present a spot price modeling method to optimize the cost-availability trade-offs for spot markets. Our goal is running jobs with similar performance to on-demand instances but at a cost near that of spot instances. We implement our control framework and evaluate it using `avrora` benchmark workload. The results show that over 95% jobs have similar performance of on-demand instances and the cost is reduced by 46% compared to using only on-demand instances.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I wish to express my sincere thanks to Dr. Bhuvan Urgaokar and Cheng Wang, for their help and support.

Chapter 1

Introduction

In recent years, as the public cloud platforms become increasingly sophisticated, the cloud providers offer a variety of virtual machine (VM) purchasing options to fulfill the needs of different tenant workloads. For example, Amazon Elastic Compute Cloud (Amazon EC2) provides three ways to pay for EC2 instances - on-demand (OD), reserved and spot. However, if a tenant does not know the pros and cons associated with different VMs, a large amount of time and money can be wasted (e.g. using only on-demand instance for batch jobs and using only spot instance for interactive jobs on EC2). In order to reduce the cost of VMs while maintaining high workload performance, a more intelligent tenant-side resource procurement method is needed.

Not surprisingly, tenant-side resource procurement has emerged as an important problem area both in research [25, 17, 8, 27, 22, 23, 6, 5, 9, 21, 4, 13] and practice. To simplify the complexity of public cloud interfaces, we break down the trade-offs among different VM types into two following basic dimensions

- **Price Dynamism:** Although the OD instances usually have static prices, the correspond-

ing spot instances (with the same hardware configurations as the OD instances) have dynamic spot prices which may change in minutes. Even though spot prices are often significantly cheaper than their corresponding OD prices, they can boost up to 10 times OD price occasionally. Furthermore, when OD instances (with OD price) are guaranteed to be available, spot instance can be terminated by cloud provider once the spot price exceeds its bid (which is set when the spot instance is launched). Several papers have explored alternate dynamic pricing designs and tenant procurement in the context of such pricing [14, 7, 12, 24, 11, 15, 18, 25, 23]. Figure 1.1 is one example of spot price of Amazon EC2.

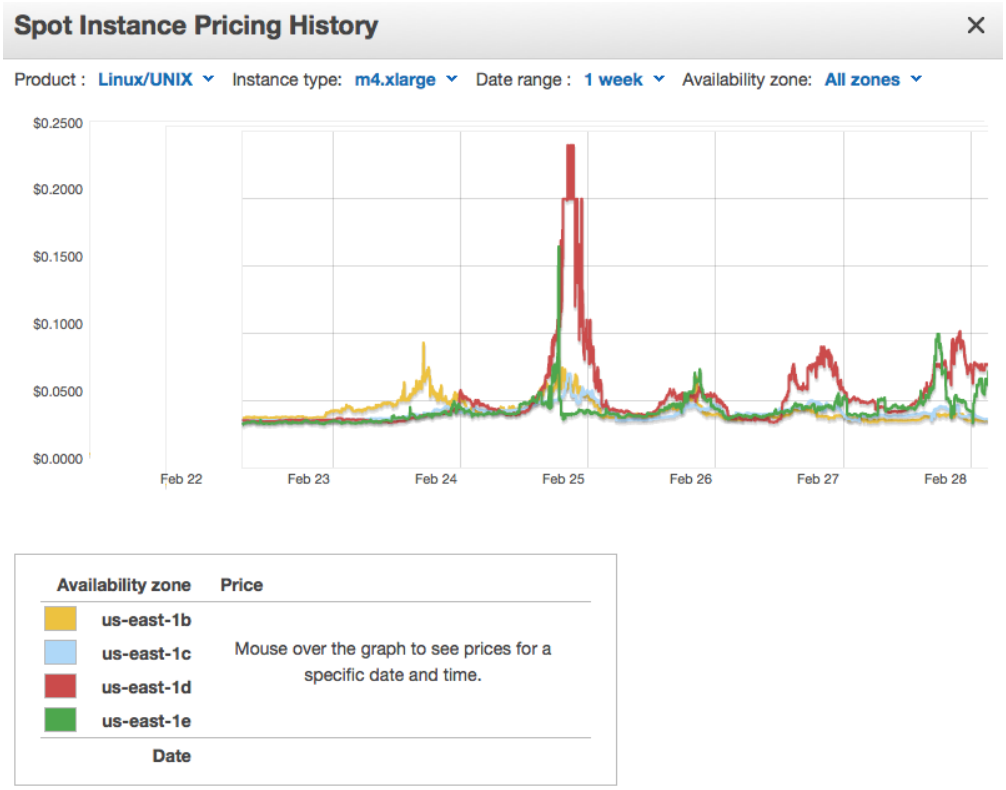
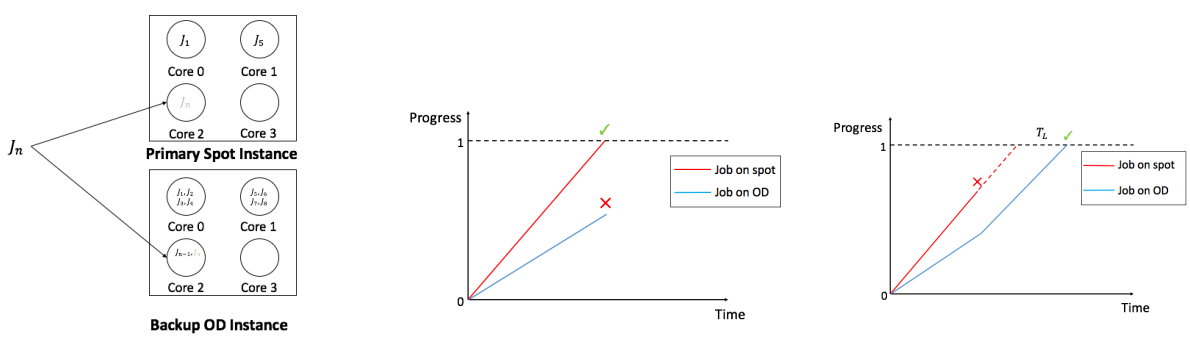


Figure 1.1: Spot price history for `m3.large` instances from Feb 22, 2015 to Feb 28,2015

- **Offer Capacity Dynamism:** A key feature of public cloud VMs that is now well-appreciated by many tenants is that there may be a discernible gap between their advertised and offered capacities. For example, Amazon EC2 burstable instances can burst above some baseline of CPU performance for a certain amount of time; Processor State Control for EC2 Instance

can increase or decrease the performance of cores (in CPU frequency). Moreover, numerous sources report that this gap can exhibit temporal variation, introducing additional complexity for tenants that wish to use such VMs [10, 19]. In this thesis, we would like to consider Amazon EC2 spot instances as an extreme form of such capacity dynamism (when a bid fails, the VM offers 0 capacity, otherwise it offers 100% capacity).

As tenants enjoy the much cheaper spot prices, they also have to take the risk of spot failures. In this case, tenants may leverage various fault-tolerance mechanisms to reduce the lost on each spot failure. For example, in [17], the authors introduce three different tolerance mechanisms for the spot market. In this thesis, we will focus on a fault-tolerance mechanism called replicating computation to specifically optimize the cost and capacity trade-offs of non-interactive batch jobs using both on-demand and spot instances. The basic idea of replicating computation is described in figure 1.2. When a job is arrived, our resource procurement controller will send it to i) a spot instance which contains up to $n\rho$ jobs at a given time and ii) an on-demand instance which contains up to $k \cdot n\rho$ jobs at a given time, where n is the number of core on the instance and $k, \rho > 0$ are some integer. If the job finishes successfully on spot instance, we will terminate the job on on-demand instance. If the spot instance fails before the job finishes, we will finish the job on on-demand instance.



(a) A new job is assigned to a spot instance and OD instance. Here, we have $n = 4, \rho = 1, k = 4$

(b) If a job finishes on spot, terminate its backup copy

(c) If a job fails on spot, boost its backup copy, where T_L is spot fail penalty

Figure 1.2: Basic idea of replicating computation

This thesis will include following topics:

- **Problem Identification:** Assume the length of each jobs T is known, given k, n, ρ and price history of all spot instances, our goal is to find a placement and bidding policy that lowers the cost while ensuring satisfactory workload performance. To achieve the goal, we need to figure out: i) which spot market should we place the job on; ii) the bid of a new spot instance if needed; iii) where to place backup copy on our backup pool.

- **Modeling:** Based on novel insights from analysis using a variety of large-scale spot price traces, we identify the contiguous period of successful bid and the average price during that period as most important and effective quantities to predict for decision making (allowing computationally scalable control design) whereas most related works focus on prediction of actual spot price or long-term statistical properties, e.g, Markovian models that may result in computationally difficult control problems (Section 2.1).

- **Framework Design and Integration:** We design a hierarchical control framework for tenants cost-effective VM procurement. At the top level, we develop a global controller periodically predicts workload properties, updates the aforementioned spot price models and computes the cost-optimal resource allocation and workload replicating decisions. At the bottom level, we deploy a local controller on each VM which allocates resources for each jobs on that VM (e.g. CPU shares).

- **Implementation on EC2:** We implement our full control hierarchy as a prototype system on EC2. On each VMs, we run each jobs on a Docker container and scale the resource allocation of container using cgroup. As a case study, we will run both artificial and real world workload on these VMs.

- **Evaluation:** To demonstrate the cost-benefit of our framework and help understand how the system might work in the real world, we carry out both trace-driven simulation with a variety of long-term spot price trace and real-world workload traces, and live experiments with artificial and real world workload to run on our EC2-based prototype system.

Chapter 2

Spot Price Modeling and Prediction

2.1 Background and Relative Work

There are a lot of research works showing that using spot instances can reduce costs. However, the lower offered capacity, which depends on tenant's bidding policy, may hurt performance. To effectively use spot instances, tenants need to predict well aspects of spot prices relevant to this cost vs. performance trade-off.

Many papers attempt to predict real-time, near-term spot prices, for example, via auto-regressive models [2]. Since spot prices could change in a few minutes, such techniques might fail to provide insights on how it might evolve in the long run. A second class of work uses a range of statistic modeling from simple empirical cumulative distributions of key parameters [17, 9] to more complex (with more memory) statistical models [26], perhaps adapting these over time. Although offering an improved treatment of longer term properties than the first class, even if updated adaptively with new observations, they discard valuable temporal information about the continuity of the spot price staying below different bid values. Therefore, they may fail to capture well the continuity of service availability, which is of great concern particularly for long-lived and stateful applications. To our knowledge, one exception is [16] wherein the sojourn time of a given discretized price value is modeled and predicted via a Semi-Markov chain. However, tenant control based on such multidimensional models would likely to suffer scalability limitations when applied to multiple spot markets with multiple bids for better availability and profitability.

2.2 Two Important Features

Rather than model the exact spot price values themselves, we argue that for purposes of capturing cost vs. performance implications, the focus should be on the two features we identify below:

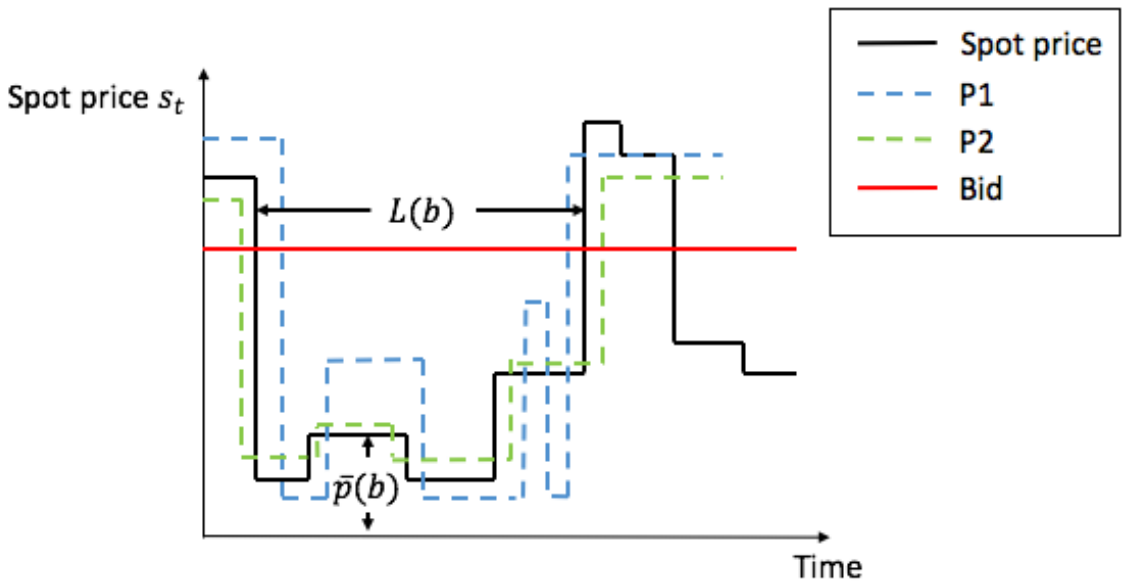
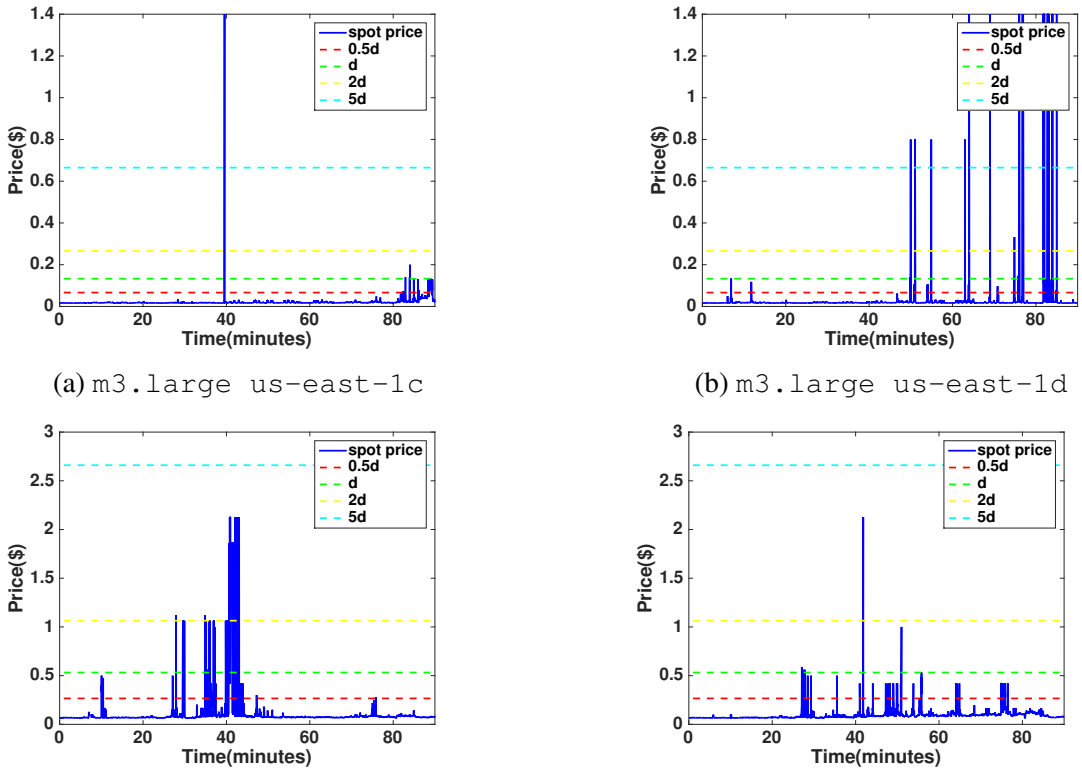


Figure 2.1: Illustrate the key idea of our spot price modeling

- **Feature I:** Since spot prices tend to be significantly smaller than on-demand prices (of comparable sized instances) during periods when a bid is successful, and since EC2 charges a tenant based on the spot price during such periods (not based on the bid placed by a tenant), attempting to predict spot prices very accurately is of little value. A visual inspection of 90-day long spot price time series from four different market in figures 2.2(a)-(d) and how these prices compare with different bid values clarifies this. In particular, it is sufficient that we predict the average spot price during such periods with reasonable accuracy. Consider the sample spot traces in figure 2.1 as an example, both two hypothetical predictors, P1 and P2, capture the average equally well but P2 captures the variance better than P1. However, since the tenants cost estimates are identical (since they have the same average value) with the two predictors, P2 is not necessarily better than P1 despite capturing the exact spot prices better.

- **Feature II:** In fact, we claim that the P1 is the preferred predictor because it is better at capturing the other important feature — the during of a successful bid. Actually, an effective predictor should not over-estimate this quantity because doing so may render a control scheme overly optimistic in its estimation of the associated cost vs. performance trade-off.



(a) `m3.large us-east-1c` (b) `m3.large us-east-1d`

(c) `m3.2xlarge us-east-1c` (d) `m3.2xlarge us-east-1d`

Figure 2.2: Four spot price time series out of several we use in our evaluation. These were collected during the 90-day period (2015-07-08 to 2015-10-06) and are chosen due to their very different properties.

2.3 Prediction and Evaluation

We analyze a large number of spot price time series collected from several market over several months [20]. We find that the key spot price features identified above tend to exhibit a form of short-term temporal locality: over relatively short timescales (a day to a few days, depending on the spot market), they tend to change little, whereas over longer timescales (weeks to months,

again depending on the spot market), they undergo more substantial changes. These observations suggest that short-term predictors of these features may perform well.

Our technique employs empirical probability distributions computed over recent sliding time windows (w most recent days) for making predictions over prediction horizons (h upcoming days). w and h must be chosen such that the aforementioned temporal locality holds. We model as a random variable $L(b)$ the length of a contiguous period during which the spot price is less than or equal to a bid b. $L(b)$ captures the lifetime of a spot instance using bid b. We denote as $\bar{p}(b) = E[p_t|L(b)]$ a random variable for the average spot price p_t during a period when the bid b is successful, which serves to estimate the cost of a spot instance procured by placing a bid b. Figure 2.1 clarifies these quantities. Large $L(b)$ and small $\bar{p}(b)$ imply long service and low costs, thereby encouraging the use of spot instances under bid b. We use a small percentile (e.g. 5th) of the recently constructed distribution of $L(b)$ - denoted as $\hat{L}(b)$ - as our prediction in the ongoing horizon. The reasoning behind this choice is that if the statistical properties of $L(b)$ do not change much between w and h, we expect that with a very high probability bid b would be successful for at least $\hat{L}(b)$ time units. We use average of $\bar{p}(b)$ during relevant w as its predictor (denoted as $\hat{\bar{p}}(b)$) during h.

Assessment Metrics: We say that an over-estimation of $L(b)$ has occurred when $\hat{L}(b) > L(b)$. This represent a scenario wherein the tenant was likely overly ambitious in using spot instances. We further define $L(b)$ over-estimation rate as the fraction of $L(b)$ predictions that result in over-estimation (denote as f). The assessment metric for $\hat{\bar{p}}(b)$ should capture the extent of its deviation from actual values. Therefore, we compute $\xi = (\bar{p}(b) - \hat{\bar{p}}(b))/\bar{p}(b)$ and define as relative deviation of $\bar{p}$ the mean value of ξ for all occurrences of $\bar{p}(b)$ in the relevant w. Lower values are better for both.

Validation: We evaluate our technique with 90-day spot traces of VM types of `m3.large` and `m3.2xlarge` in availability zone `us-east-1c` and `us-east-1d`. These are also the traces shown in figure 2.2 (a)-(d). We employ $w = 7$ days and $h = 1$ day. For each trace, we pick bid price $b \in \{0.5d, d, 2d, 5d\}$, where d is the corresponding OD price. Table 1 represents our assessment metrics for 16 representative traces. We highlight entries where $f > 15\%$ or $\xi > 30\%$

(i.e. poor predictions).

instance-type	c3.large		r3.large		m3.large		m3.2xlarge	
metric	f	ξ	f	ξ	f	ξ	f	ξ
market 1: us-east-1b								
$0.5d$	0.0947	0.0703	0.0756	0.2900	0.1607	0.0844	0.0991	0.1617
$1d$	0.0698	0.0698	0.0927	0.2392	0.1406	0.0855	0.0955	0.1678
$2d$	0.0690	0.0701	0.0802	0.4256	0.1240	0.0937	0.0971	0.3484
$5d$	0.0698	0.0714	0.0714	0.5194	0.1364	0.1129	0.0417	0.2966
market 2: us-east-1c								
$0.5d$	0.1014	0.0586	0.0519	0.3460	0.1224	0.0796	0.0823	0.1052
$1d$	0.1238	0.1120	0.0509	0.3567	0.0706	0.0689	0.0672	0.1050
$2d$	0.1456	0.1097	0.0858	0.7027	0.0244	0.0836	0.0755	0.1657
$5d$	0.1373	0.1272	0.0728	1.1451	0.0244	0.0836	0.0000	0.4896
market 3: us-east-1d								
$0.5d$	0.0920	0.0553	0.0696	0.0142	0.1024	0.0619	0.1301	0.1414
$1d$	0.0482	0.0725	0.0861	0.1990	0.1045	0.0727	0.0909	0.1602
$2d$	0.0482	0.0725	0.0968	0.5204	0.0992	0.0989	0.0244	0.1327
$5d$	0.0482	0.0729	0.0897	0.8901	0.1327	0.0907	0.0000	0.1354
market 4: us-east-1e								
$0.5d$	0.0244	0.0574	0.0857	0.1441	0.0417	0.1659	0.0957	0.0692
$1d$	0.0241	0.0554	0.0673	0.1565	0.0469	0.1652	0.0000	0.1073
$2d$	0.0241	0.0554	0.0642	0.1577	0.0296	0.1787	0.0000	0.1073
$5d$	0.0241	0.0562	0.0741	0.1609	0.0000	0.4128	0.0000	0.1073

Table 2.1: Evaluation of our prediction technique for four different markets and four different advertized instance sizes

Chapter 3

Algorithm Design

In this chapter, we will describe how we design our replicating computation method and resource procurement strategies using model we describe in chapter 2. Section 3.1 shows the blueprint of our control framework and section 3.2 shows our basic strategy for procuring resources and placing jobs. At the end of this chapter, we also discuss some heuristic method we might use to boost jobs when spot fails.

3.1 Overview

We design a hierarchical control framework, as shown in figure 3.1, to optimize our tenant resources procurement and job placement. In the upper level, we have a scheduler and an optimizer which are responsible for procuring instances and placing jobs to instances we procured from EC2. In the lower level, we have a job manager which allocate system resources for jobs on each instance we own. We will describe basic functions of these parts in following paragraphs.

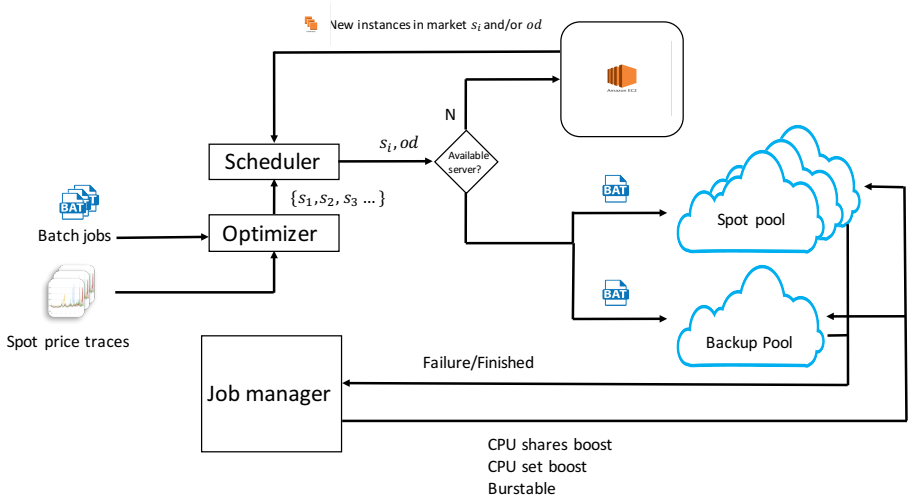


Figure 3.1: Overview of framework design

Optimizer. The optimizer receives job request asynchronously and reads last w-days prices of all spot markets from EC2 periodically (e.g. 1 minute). Then it generates a list of feasible markets using methods we will discuss in 3.2 and sends it to the scheduler.

Scheduler. The schedule picks a market s from the list it receives from optimize and check if available servers exist in market s and backup pool (note that the backup pool itself is a market, which we denote as od). If yes, the scheduler runs a primary copy of the job on spot server and a backup copy of the job on backup server simultaneously. If not, the scheduler grab new spot/od instances from Amazon EC2 then repeat the processes.

Job manager. The job manager monitor each jobs on spot and backup instances. If a job finishes on spot instance, job manager terminates its backup copy on backup instance. If a job fails on spot instance, job manager boosts its backup copy on backup instance.

3.2 Job Placement Algorithm design

In order to optimize the cost and performance trade-offs, our job placement algorithms needs to solve the following two problems:

1. **Placement of primary copy:** which spot market and what bid should we choose to place primary copy of the arriving job?

2. **Placement of backup copy:** which backup slot/vCPU should we choose to place backup copy of the arriving job?

Fortunately, based on spot price model we discuss in chapter 2, these problems could be easily solved. Denote a spot group g as a 2-tuple $g = (s, b)$, where s is spot market and b is bid. After reading spot price histories from Amazon EC2, the optimizer can compute $L_{s_i}(b_m)$ and $\bar{p}_{s_i}(b_m)$ for each EC2 market s_i and $b_m \in B$ which is pre-define by tenant. Upon a new job j arrive, the optimizer analyze the length of j, which we denote as l_j, and compare it with $L_{s_i}(b_m)$. The optimizer then append $g = (s_i, b_m)$ to a list if $L_{s_i}(b_m) >= l_j$ and sort the list based on $\bar{p}_{s_i}(b_m)$. After all of operations, the optimizer should return a list of market $g_1, g_2, g_3, ..., g_k$ such that $L_{g_i.s} >= l_j$ for all $1 \leq i \leq k$ and $\bar{p}_{g_1.s}(g_1.b) \leq \bar{p}_{g_2.s}(g_2.b) \leq \cdots \leq \bar{p}_{g_k.s}(g_k.b)$. The pseudo-code for the optimizer is stated as follow:

Algorithm 1 Optimizer

```
function OPTIMIZER(job, spotMarketList, B)
    feasibleGroupList ← ∅
    for s in spotMarketList do
        for b in B do
            if s.L(b) >= job.length then
                feasibleGroupList.append((s, b))
            end if
        end for
    end for
    Sort feasibleGroupList based on p̄(b)
    return (job, feasibleGroupList)
end function
```

According to table 2.1, our algorithm guarantees that up to 98% probability the market in $feasibleGroupList$ will not fail before the job j finishes. The optimizer then send the list and the job j to the scheduler. Upon receiving the list and job, the scheduler needs to pick a group g from the list so that the expect cost $\frac{l_j \cdot \bar{p}_{g.s}(g.b)}{N_P}$, where N_P is the maximum number of job running on spot servers, is minimized (i.e. pick the group g with minimum $\bar{p}_{g.s}(g.b)$). Since the list is sorted, we can directly pick the first element from the list. Subsequently, we need to find a server in the chosen group with available slot (e.g. vCPU) and run the job on that slot. Since we give every job a full core on spot instances (i.e. $\rho = 1$), adding jobs on spot instance will not affect the performance of other jobs. For $\rho > 1$, we will use the method of finding slots on backup instances, which we will discuss in next paragraphs, to minimize the performance impact to other jobs.

After running primary copy on spot instance, we also need to run a backup copy on on-demand instance. Unlike spot instances, we only have one market for backup instances. Thus, market picking problem in this case is trivial. However, since we run $N_B = k\rho$ jobs per core on backup instances, adding jobs to backup instances might slow down the processes of other jobs. In this case, upon a new job arrive, we greedily run it on the slot/vCPU with most CPU capacity (e.g. least number of job running on the CPU) so that the impact to other running jobs is minimized.

Now, we have solved the two problems we state at the beginning of this section. However, there is not always an available slot/instance in eight spot pool nor backup pool. In this case, we need to launch new spot/backup instances from the EC2 market and bid we choose. Algorithm 2 shows the pseudo-code for the scheduler.

According to algorithm 2, our scheduler will always choose the feasible group with lowest $\bar{p}(b)$ and run all jobs on that group. However, this could very dangerous. If we run all jobs on one group, once the group fails, all of existing jobs will be failed, which will result in a huge failure latency and cost. To avoid this, instead of always picking the group with the lowest $\bar{p}(b)$ value, we now randomly choose a group from groups with lowest n $\bar{p}(b)$ value. By doing this, our job will distribute on different groups instead of one, which reduce the probability that all jobs fail at the same time. Algorithm 3 shows the pseudo-code for randomized version of the scheduler.

Algorithm 2 Scheduler

```
function SCHEDULER(job, feasibleGroupList, odMarket)
    g ← feasibleGroupList[0]
    if g has no available instance then
        launch spot instance in group g from Amazon EC2
    end if
    if odMarket has no available instance then
        launch on-demand instance from Amazon EC2
    end if
    spServer, spSlot ← available server and slot from g
    odServer, odSlot ← available server and slot with maximum CPU capacity from odMarket
    Run job on (spServer, spSlot, odServer, odSlot)
end function
```

Algorithm 3 Scheduler-Randomized

```
function SCHEDULER-RANDOMIZED(job, feasibleGroupList, odMarket, n)
    index ← random integer from 0 to n − 1
    g ← feasibleGroupList[index]
    if g has no available instance then
        launch spot instance in group g from Amazon EC2
    end if
    if odMarket has no available instance then
        launch on-demand instance from Amazon EC2
    end if
    spServer, spSlot ← available server and slot from g
    odServer, odSlot ← available server and slot with maximum CPU capacity from odMarket
    Run job on (spServer, spSlot, odServer, odSlot)
end function
```

3.3 Heuristic Improvements

Even though our spot price model guarantees a high availability for each spot instance, it might still fails sometime. In this case, the failed jobs will finish on backup instances, which have lower performance because of core-sharing. In order to improve the performance of failed job, we design soem heuristic to boost failed jobs by sacrificing performance of not-failed jobs on backup instance.

3.3.1 CPU shares boost

The CPU shares indicates the amount of CPU resource a job can use. The maximum amount of CPU utilization of a job can be computed by

$$c = \frac{\text{CPU shares of the job}}{\text{total CPU shares of jobs on the core}}$$

For example, we have two jobs j_1, j_2 with CPU shares 256 and 1024, the maximum amounts of CPU each job can use are

$$c_{j_1} = \frac{256}{256 + 1024} = 0.2$$
$$c_{j_2} = \frac{1024}{256 + 1024} = 0.8$$

When no job fails, every job on backup instances has equal low CPU shares value (e.g. 256) and evenly shares the CPU. However, when job manager detects a job failure, it sets the CPU shares of failed job's backup copy to a higher values (e.g. 1024). Consequently, the failed-jobs occupy more CPU resources and not-failed jobs occupy less CPU resources.

3.3.2 CPU sets boost

Although CPU shares boost allows us boosting failed-jobs effectively, it never allows the failed-job occupy a full core if the more than one jobs running on the core. Fortunately, we can break this limitation using CPU sets boost method and boost the job even further.

CPU sets provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks. For example, suppose we have 4 jobs $\{j_1, j_2, j_3, j_4\}$ running on a system with 2 cores and job j_1, j_2 run on the first core and j_3, j_4 run on the second core. In this case, the CPU sets of j_1, j_2 is 0 and CPU sets of j_3, j_4 is 1. Suppose further that these jobs share the CPU evenly and each of them occupies 50% of a full core. Now, suppose the job manager detects j_4 fails. In this case, it change the CPU set of j_3 to 0 and left the CPU set of j_4 unchanged. Therefore, j_1, j_2, j_3 run on and evenly share the first core and j_4 occupy the entire second core now.

Chapter 4

Implementation and Evaluation

In this chapter, we will illustrate how we implement the control framework we describe in section 3 and how effective is our control framework. Section 4.1 shows the implementation details of different parts of our control framework. Section 4.2 evaluate the performance and cost-efficacy of our control framework under long-term simulation and short-term real-world experiment. At the end of this chapter, we also point out some possible enhancements which we will investigate in the future.

4.1 Implementation

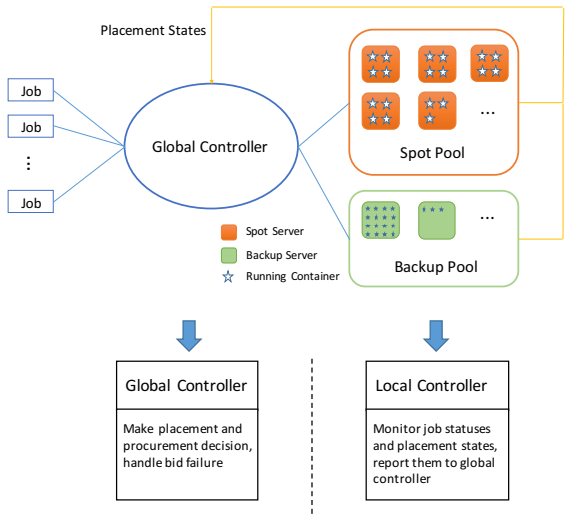


Figure 4.1: Architecture of our control framework

In this section, we will describe implementations of different parts of our control framework. Figure 4.1 shows the architecture of our control framework. We implement this framework using Python programming language and Docker container [1] with `cgroup` to allocate system resources. We will describe two main parts, global controller and local controller, in following two subsections.

4.1.1 Global Controller

The *global controller* receives job requests from users then do the the following things:

1. Analyze the job length, say j.

2. Find a feasible slot (e.g. $L(b) > j$ and number of existing jobs less than maximum number of jobs on the server) from existing spot servers and run the job on that server.

3. Find an open slot from existing backup servers and run the job on that server.

4. If there is no such feasible spot slot, pick the best EC2 market based on current $L(b)$ and $\bar{p}(b)$ (e.g. market with smallest $\bar{p}(b)$ and $L(b) > j$) and request a new server with bid b such that $L(b) > j$ from that EC2 market. Then run the job on the new server.

5. If there is no open backup slot, request a new on-demand server from EC2.

Moreover, the global controller periodically check placement states of jobs (e.g. which job on which core, CPU shares etc.) on existing servers. If a server finish all of its jobs, it would be considered as "idle" and terminated by global controller. If a spot server is revoked, global controller will boost corresponding backup copy of fail jobs by increasing the their CPU shares on backup servers (e.g. from 256 to 1024).

A summary of classes and methods of the global controller can be found at appendix A.1.

4.1.2 Local Controller

The $local\ controller$ runs on each EC2 servers and continuously receives requests from global controller. The main functions of local controller are the following:

1. **Run new jobs.** After receiving a run-job request from global controller, the local controller will create a docker container and run it with specific commands, CPU set and CPU shares provided by global controller. Then the local controller will send the new container ID to the global controller.

2. **Stop existing jobs.** This will only happen on backup servers. Once a job is finished on spot server, the global controller will send a stop request to the server running the backup copy of the job. After local controller receiving this request, it will stop the containers with IDs provided by global controller and reallocate the resources for other jobs.

3. **Change control groups of jobs.** Local controller allows the global controller to change CPU set (i.e. which cores the containers run on) and CPU shares (i.e. the amount of CPU resources the containers can use) of containers.

4. **Report status.** Local controller periodically sends the placement states (e.g. CPU shares, CPU set, number of jobs) on the server to the global controller, which will use these information to decide who is the best server to place next job.

Additionally, local controller also has auxiliary functions, such as importing images to docker and exporting file system of a container, to allow users run arbitrary jobs on the server and save arbitrary results. A summary of methods of local controller can be found at appendix A.2.

4.2 Evaluation

In this section, we will demonstrate the effectiveness of our control framework using `avrora` benchmark in DaCapo benchmark suite[3] as workload. Firstly, in section 4.2.1, we will describe our experiment setup and some baselines for comparison. Second, in section 4.2.2, we conduct a simulation by scaling real-world three-month spot price traces from Amazon EC2 and demonstrate the cost-efficacy of our control framework in the long run. Finally, in section 4.2.3, we run our control framework implementation on EC2 with 24-hour spot price traces to demonstrate short-term performance of our control framework.

4.2.1 Experiment Setup

Spot market. In this experiment, we use `m3.xlarge` instance type in availability zone `us-east-1c` and `us-east-1d`. The spot price traces of these two market are shown in Figure 4.2. Moreover, we use bid $= \{p^{od}, 5p^{od}\}$ for each of market, where $p^{od} = 0.266$ is the on-demand price of `m3.xlarge`. Conclusively, writing instance group as a 3-tuple $s_i = (\texttt{type}, \texttt{zone}, \texttt{bid})$,

we have following 5 types of instance group:

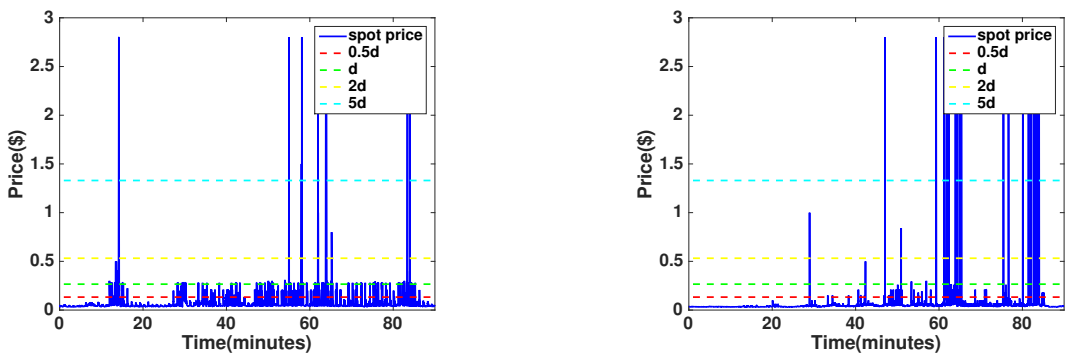
$$s_1 = (\texttt{m3.xlarge}, \texttt{us-east-1c}, p^{od})$$

$$s_2 = (\texttt{m3.xlarge}, \texttt{us-east-1c}, 5p^{od})$$

$$s_3 = (\texttt{m3.xlarge}, \texttt{us-east-1d}, p^{od})$$

$$s_4 = (\texttt{m3.xlarge}, \texttt{us-east-1d}, 5p^{od})$$

$$s_{od} = (\texttt{m3.xlarge}, \texttt{On-Demand}, 0)$$



(a) `m3.xlarge us-east-1c` (b) `m3.xlarge us-east-1d`

Figure 4.2: Spot traces use for evaluation

Workload. We generate our workload using exponential distribution with rate $\lambda = \frac{1}{6}$ (i.e. expect 10 jobs come per hour). The length of each job depends on the total iteration count n, which is generated randomly, of `avrora` benchmark.

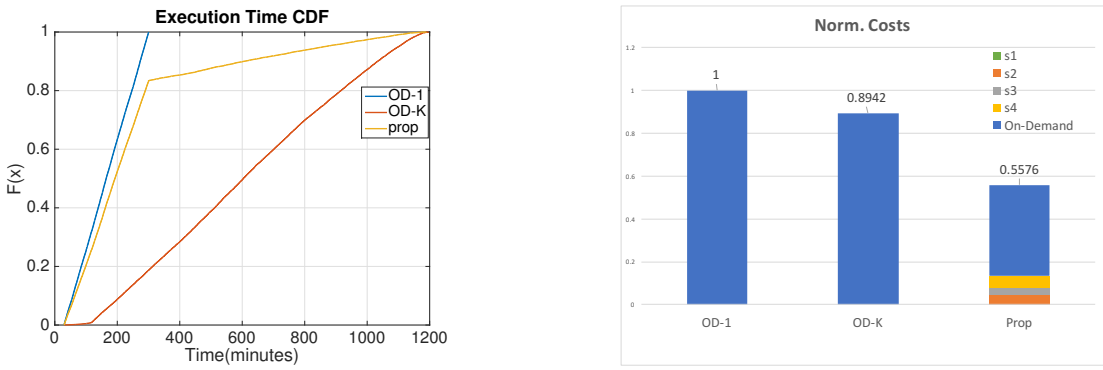
Baselines. We denote "prop" as our replication approach. During the experiment, we will compare our approach will following two baselines:

1. **OD-1:** Run all jobs on On-Demand instance, 1 job per core.

2. **OD-K:** Run all jobs on On-Demand instance, k jobs per core.

4.2.2 Simulation

We simulate our replicating approach using 90-day spot price traces from Amazon EC2 and compare it with our baselines mention in section 4.2.1 with $k = 4$. The length of jobs in this simulation are randomly chosen between 30 and 300 minutes. Figure 4.3 shows the results of our simulation.



(a) (b)

Figure 4.3: (a) shows the execution time CDF of two baselines and our prop approach. (b) compares the cost breakdown of our baselines and prop approach

According to the results, we have several observations. First of all, our replicating approach is able to save as much as 45% total costs compares to OD-1 baseline and 38% total costs compares to OD-K baselines. This is because OD-1 always put jobs on on-demand instances even if the $L(b)$ of a much cheaper spot market is long enough. Additionally, even though OD-K put more jobs on each server, it also worse the throughput of each server. Thus, jobs are run much slower on these server, which leads to more cost.

Second, 83% of prop jobs have similar execution time with OD-1 jobs, which have the best performance. This is because jobs run on spot instances enjoy the same computational resources (e.g. CPU) as jobs run on on-demand instances in OD-1. Even though the jobs may fail on spot instances, our controller will boost the fail jobs on backup servers so that they will not slow down too much.

Third, our prop approach choose multiple spot market to run the jobs. When current market

fails, it will immediately find another feasible market to run new jobs. In fact, since we only use 4 instance groups in this experiment, there are some periods of time such that there is no feasible market (i.e. spot prices exceed maximum bid for all markets). Thus, if we add more spot markets, we can expect that the probability of all market fails will decreases and our prop approach will be more effective.

4.2.3 Cost-efficacy on EC2

To demonstrate the effectiveness of our control framework in real-world scenario, we cut a 24-hour spot price trace from traces we show in figure 2.2c and figure 2.2d respectively and run our control framework with `avrora` benchmark workloads generated using method described in section 4.2.1.

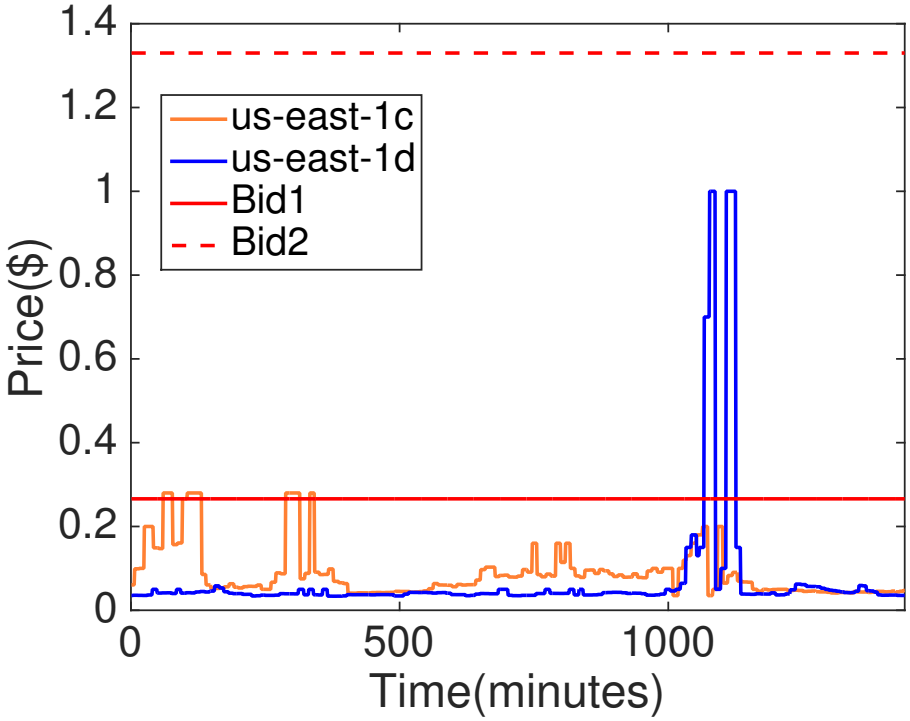


Figure 4.4: 24-hours spot price traces used for evaluation

Figure 4.4 shows the spot prices traces we use in this experiment. Note that we intentionally choose the traces with spot failure so that we can observe the failure operation of our control

framework.

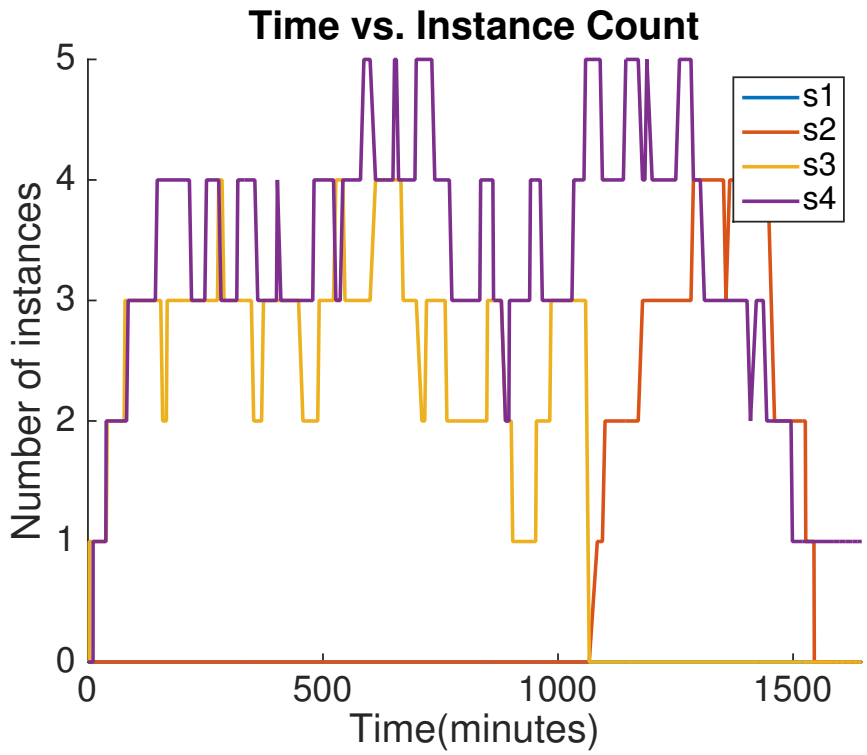
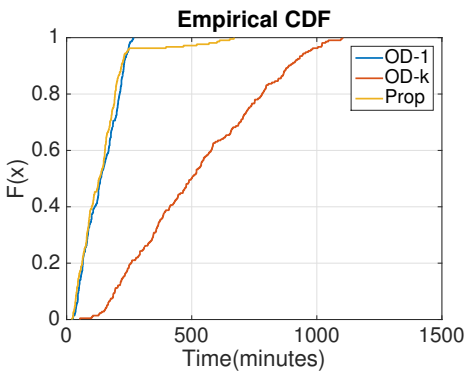
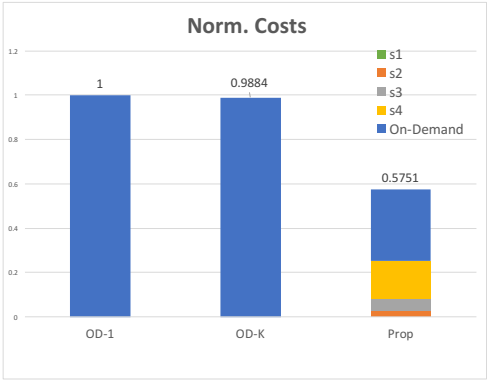


Figure 4.5: Spot usage during 24-hours experiment

Figure 4.5 shows the spot usage during the 24-hours experiment. We can see at the beginning of this experiment, the global controller choose market s_3 and s_4 to run jobs because market s_3 and s_4 have smaller $\bar{p}(b)$ than market s_1 and s_2. However, at around 1050 minute, all instances in market s_3 are terminated because of bid failure of market s_3. In this case, the global controller automatically choose market s_2 to take the place of market s_3. The not-failing market, s_4 is not affected.



(a)



(b)

Figure 4.6: (a) shows the execution time CDF under different strategies (b) compares the cost breakdown under different strategies

Figure 4.6 shows the results of our 24-hours experiment. Based on the results, we have the following observations. First of all, the CDF graph is similar with what we get form our simulation. However, our control framework works even better than what we expect in short-term real-world scenario (i.e. over 95% of prop jobs have similar performance with OD-1 jobs compares to 83% in our simulation). Secondly, the cost-saving is 46% in this experiment, which is also similar to the cost-saving in our long-term simulation.

In conclusion, while guaranteeing success of each job, our control framework is able to achieve up to 46% overall cost-saving and maintain high performance. Another observation from our experiment is that, even though we use spot instances as our primary server, most of costs come from on-demand instances. This means we can achieve even better cost-efficacy by adjusting the k value (e.g. use less on-demand instances by ”squeezing” more jobs on one backup server). We will investigate this further in our future works.

Chapter 5

Conclusion

Our spot price modeling approach and replicating method allow over 95% of jobs running with performance of on-demand instances and 46% lower costs. However, our current work only considers using on-demand instances and regular types of spot instances. As part of our future work, we plan to i) use burstable EC2 instances, which are a new low-cost, general purpose instance type that provide a baseline level of CPU performance with the ability to burst above the baseline, as our backup instances; ii) use EC2 reserved instances, which allow tenants reserve instances with a period of time in exchange for a significant discount (up to 75%) compared to On-Demand instance pricing.

Bibliography

[1] Docker. `https://www.docker.com/`.

[2] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. Deconstructing amazon ec2 spot instance pricing. In *Proc. CLOUDCOM* (2011).

[3] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, Oct. 2006), ACM Press, pp. 169–190.

[4] CHENG, Y., IQBAL, M. S., GUPTA, A., AND BUTT, A. R. Cast: Tiering storage for data analytics in the cloud. In *Proc. ACM HPDC* (2015).

[5] CHOHAN, N., CASTILLO, C., SPREITZER, M., STEINDER, M., TANTAWI, A., AND KRINTZ, C. See spot run: Using spot instances for mapreduce workflows. In *Proc. USENIX HotCloud* (2010).

[6] CONLEY, M., VAHDAT, A., AND PORTER, G. Achieving cost-efficient, data-intensive computing in the cloud. In *Proc. ACM SOCC* (2015).

[7] FENG, Y., LI, B., AND LI, B. Bargaining towards maximized resource utilization in video streaming datacenters. In *Proc. of INFOCOM* (2012).

[8] HE, X., SHENOY, P., SITARAMAN, R., AND IRWIN, D. Cutting the cost of hosting online services using cloud spot markets. In *Proc. ACM HPDC* (2015).

[9] JAVADI, B., THULASIRAMY, R. K., AND BUYYA, R. Statistical modeling of spot instance prices in public cloud environments. In *Proc. IEEE UCC* (2011).

[10] JAYATHILAKA, H., KRINTZ, C., AND WOLSKI, R. Response time service level agreements for cloud-hosted web applications. In *SoCC'15* (2015).

[11] KAUSHIK, R. T., SARKAR, P., AND GHARAIBEH, A. Greening the compute cloud's pricing plans. In *Proc. HotPower* (2013).

[12] LI, H., WU, C., LI, Z., AND LAU, F. Profit-maximizing virtual machine trading in a federation of selfish clouds. In *Proc. of IEEE INFOCOM* (2013).

[13] MENACHE, I., SHAMIR, O., AND JAIN, N. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *Proc. USENIX ICAC* (2014).

[14] NIU, D., FENG, C., AND LI, B. Pricing cloud bandwidth reservations under demand uncertainty. In *Proc. of ACM SIGMETRICS* (2012).

[15] SHARMA, B., THULASIRAM, R. K., THULASIRAMAN, P., GARG, S. K., AND BUYYA, R. Pricing cloud compute commodities: A novel financial economic model. In *Proc. IEEE/ACM Ccgrid* (2012).

[16] SONG, Y., ZAFER, M., AND LEE, K. Optimal bidding in spot instance market. In *Proc. of IEEE INFOCOM* (2012).

[17] SUBRAMANYA, S., GUO, T., SHARMA, P., IRWIN, D., AND SHENOY, P. Spoton: A batch computing service for the spot market. In *Proc. ACM SOCC* (2015).

[18] WANG, C., NASIRIANI, N., KESIDIS, G., URGAONKAR, B., WANG, Q., CHEN, Y., GUPTA, A., AND BIRKE, R. Recouping energy costs from cloud tenants: tenant demand response aware pricing design. In *ACM e-Energy* (2015).

[19] WANG, C., URGAONKAR, B., GUPTA, A., CHEN, L. Y., BIRKE, R., AND KESIDIS, G. Effective capacity modulation as an explicit control knob for public cloud profitability. Tech. rep., http://www.cse.psu.edu/research/publications/tech-reports/2016/CSE-16-001.pdf, 2016.

[20] WANG, C., URGAONKAR, B., GUPTA, A., AND LIANG, Q. Navigating price v. offered capacity v. scaling granularity trade-offs in the cloud: A memcached case study combining on-demand and spot instances. Tech. rep., CSE-16-002, Penn State University. `http://www.cse.psu.edu/~bhuvan/CSE-16-002.pdf`, 2016.

[21] WANG, W., LI, B., AND LIANG, B. To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds. In *Proc. USENIX ICAC* (2013).

[22] WEN, J., LU, L., CASALE, G., AND SMIRNI, E. Less can be more: Micro-managing vms in amazon EC2. In *Proc. IEEE CLOUD* (2015).

[23] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. ACM SOSP* (2013).

[24] XU, H., AND LI, B. Maximizing revenue with dynamic cloud pricing: the infinite horizon case. In *Proc. of IEEE ICC* (2012).

[25] XU, Z., STEWART, C., DENG, N., AND WANG, X. Blending on-demand and spot instances to lower costs for in-memory storage. In *Proc. IEEE INFOCOM* (2016).

[26] ZAFER, M., SONG, Y., AND LEE, K.-W. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing* (Washington, DC, USA, 2012), CLOUD '12, IEEE Computer Society, pp. 75–82.

[27] ZHENG, L., JOE-WONG, C., TAN, C. W., CHIANG, M., AND WANG, X. How to bid the cloud. In *Proc. ACM SIGCOMM* (2015).

Appendices

Appendix A

Summary of Controllers

This chapter will summary the classes and methods we implemented in the global and local controller.

A.1 Global Controller

Classes: Instance

A class store information about an EC2 instance.

Attributions	
`string ip`	IP address of this instance
`boolean is_spot`	If this is a spot instance
`int num_core`	The number of cores this instance has
`float Lb`	The current $L(b)$ of this instance
`float Es`	The current $\bar{p}(b)$ of this instance
`float bid`	The bid of this instance
`obj created_time`	The created time of this instance
Methods	
`void report()`	Print status of this instance
`boolean is_idle()`	Return true if no job is running on this instance
`void stop_all()`	Stop all containers on this server
`int find_open_slot()`	Return a CPU set number if it is available

Table A.1: Summary of class Instance

Classes: Job

A class store information about a job.

Attributions	
`string spot_ip`	The IP address of the spot server running this job
`int spot_cpuset`	The spot CPU set number of the core running this job
`string backup_ip`	The IP address of the backup server running this job
`int backup_cpuset`	The backup CPU set number of the core running this job
`string image`	The container image ID of this job.
`string cmd`	The command run in the container.
`boolean spot_fail`	If the spot copy is fail
`string spot_cid`	The ID of container running this job on spot server
`string backup_cid`	The ID of container running this job on backup server
`float job_length`	The length of this job in minutes
`int job_id`	The ID of this job
Methods	
`void run(obj start, int backup_cpu_shares)`	run the job on both spot and backup servers
`void set_backup_cpushares(int cpu_shares)`	Set this job's CPU share on backup server to specific value
`void update()`	Check if this job is finished. If it is finished on spot, terminate its corresponding backup copy

Table A.2: Summary of class Job

Classes: ec2Market

A class store information about a EC2 market.

Attributions	
`string instance_type`	The type of the instances
`Instance[] active_list`	Existing servers list
`float bid`	Bid for this market
`float Lb`	$L(b)$ for this market given bid b
`float Esb`	$\bar{p}(b)$ for this market
`float spot_price`	Spot price for this market
Methods	
`void add_instance(obj time)`	Request a new instance from this market
`void remove_active(time)`	Remove all active instances
`void check_spot_fail()`	Check if spot instances in this market are fail, if yes, remove all active instances
`void status()`	Report statuses of all active instances in this market
`void remove_idle()`	Remove all idle instances in this market

Table A.3: Summary of class ec2Market

Summary of Methods

`get_vm(num_spot, num_backup)`	Return the specific number of new spot and backup server lists

Table A.4: Summary of methods in global controller

A.2 Local Controller

Summary of Methods

`mount_fs(dev_name, dst)`	Mount a file system from Amazon EBS
`run_job(image_name), cmd, cpu_set, cpu_shares=1024`	Run a job on docker container
`import_image(filename)`	Import an image to docker from file
`export_fs(container_id, dst_filename)`	Export file system of a container
`remove_containers(container_ids)`	Remove specific containers
`stop_containers(container_ids)`	Stop specific containers
`check_status(ids=None)`	Check status of container in this server
`check_termination()`	Check if this instance is terminating by EC2
`set_cpu_shares(container_id, weight)`	Set CPU shares of a container to specific value
`set_cpuset_cpus(container_id, core)`	Set a container to specific CPU set

Table A.5: Summary of methods in local controller

Academic Vita

Qianlin **Liang**

3511 Plaza Drive, State College, Pennsylvania, 16801

(814) 777-1518 | Qianlin.Liang@hotmail.com

Education

The Pennsylvania State University, Schreyer Honors College *University Park, PA*

B.S. in Computer Science, minor in Mathematics *Aug. 2012 - May. 2016*

- Dean's list all semesters

Skills

Programming	Python, C/C++, JAVA, JavaScript, HTML5, CSS, MATLAB, Bash, LaTeX
Software	MATLAB, Mathematica, Vim, Git, Microsoft Visual Studio, VMware Fusion/Workstation
Operating Systems	UNIX/Linux, OS X, Windows

Experience

Undergraduate Research, Computer Systems Lab *University Park, PA*

Student Researcher *May 2015 - PRESENT*

- Analyzed Amazon EC2 Spot market history price and developed statistic model to predict EC2 Spot market price
- Implemented controller to launch, terminate and run jobs on EC2 instances programactically using Python
- Designed and implemented algorithm for EC2 users to lessen their cost while maintaining high reliability

Remcom Wireless InSite Web Interface *State College, PA*

Learning Factory Project Sponsored by Remcom, Inc. *Aug. 2015 - Dec. 2015*

- Implemented a web interface for Remcom's Wireless Insite, a wireless EM propagation software, using Node.js and React framework
- Developed application for converting origin Wireless InSite data to a common data-interchange format
- Cooperated with Remcom's engineering team to design and debug the project through GitHub

CRUD Device Driver *University Park, PA*

Class Project *Aug. 2014 - Dec. 2014*

- Implemented a user-space device driver for a file-system built on top of an object storage device using C language
- Modified and extended the device driver to communicate with the CRUD device over a network using TCP/IP protocol
- Tested and debugged the device driver using GDB tools

Employee Management & Appointment Scheduling System *University Park, PA*

Class Project *Jan. 2014 - May 2014*

- Implemented a GUI-based program to manage unlimited size of employee and schedule of appointment using Java
- Established a Derby database to store employee information and schedules

Student Gradebook Management System *University Park, PA*

Class Project *Aug. 2013 - Dec. 2013*

- Designed and implemented a program to manage unlimited size of student and their grades using C++
- Designed using object-oriented programming and implemented by using efficient algorithms and data structures

Honors & Awards

2013	**The President's Freshman Award**, Undergraduate Scholastic Awards	*University Park, PA*
2014	**The President Sparks Award**, Undergraduate Scholastic Awards	*University Park, PA*