Smart Spot Instances for the Supercloud

Qin Jia Zhiming Shen Weijia Song Robbert van Renesse Hakim Weatherspoon

Cornell University

Abstract

In this paper, we explore the use of live VM migration to take advantage of spot markets such as provided by Amazon and Google. These markets provide an exciting low cost alternative to regular VM instances, but the threats of price spikes and premature termination severely limit their usability. Migration can address these threats: spot market instances facing price hikes or termination can migrate to other instance types, including regular ones. Reliability can be further improved by replication. In this paper we investigate various design options and present some preliminary results of experiments with dynamic programming techniques, both using simulation and using a realistic deployment. We find that in unstable markets we can achieve significant savings at low overhead and while maintaining good reliability.

1. Introduction

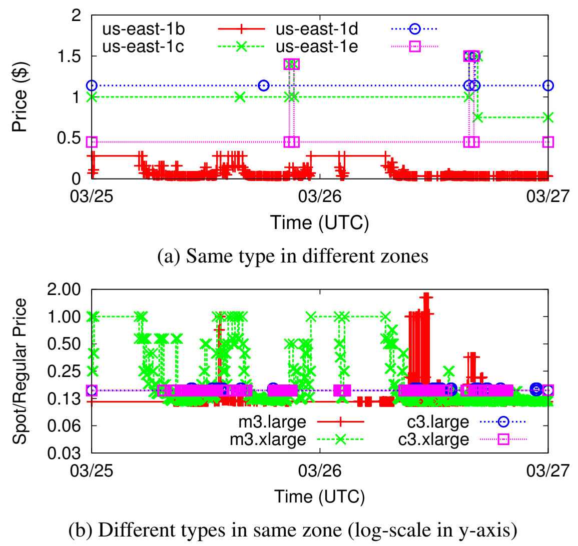
The Supercloud is an Openstack cloud that does not run on a dedicated cluster, but instead runs on resources allocated in various other clouds, including Amazon EC2, Microsoft Azure, Google Compute Engine, and private clouds (Jia et al. 2015). The Supercloud goes beyond federated clouds in that it supports management operations fully, including migration between availability zones and even autonomous and heterogeneous clouds. The Supercloud implementation leverages nested virtualization and Software Defined Networking, and achieves good performance and low overhead.

In this paper we explore the use of Supercloud migration in order to make the Spot Instances or Preemptible Instances as provided by Amazon and Google more attractive. For example, Amazon EC2 provides a *Spot Market* that allows users to take advantage of underutilized compute resources. *Spot Instances* perform the same as On-Demand Instances and are often much cheaper. But Spot Instances present risks, as described below.

First, Spot Instance prices can change rapidly—sometimes every 5 minutes, and occasionally the price can significantly exceed the price of an On-Demand Instance. Figure 1a shows the price history of the `m3.xlarge` instance type from March 25 to March 27, 2015 in different availability zones in the Amazon Virginia region.[1] The prices change (seemingly) independently in different availability zones. In each availability zone, the price can either change rapidly and in unpredictable ways (`us-east-1b`) or be quite stable (`us-east-1d`). It has been shown that the price changes have no correlation to the historical price (Mazzucco and Dumas 2011).

[1] Our Amazon account does not grant access to the `us-east-1a` zone.

CrossCloud'16 April 18, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4294-0/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2904111.2904114



(a) Same type in different zones

(b) Different types in same zone (log-scale in y-axis)

Figure 1. Spot Instance Price History

Second, unlike traditional On-Demand Instances, Spot Instances can be terminated at any time by EC2 (in which case Amazon does not bill for the interrupted hour). To use Spot Instances, users specify a maximum bid price when starting the instances. Spot Instances are charged hourly at the beginning of an instance hour. The Spot Instances will be terminated automatically when the Spot price exceeds the maximum bid. In this case, Amazon sends out a termination notice and gives a two-minute grace period, during which an emergency recovery operation can be performed.

This unpredictable termination limits how Spot Instances can be used effectively. When using Spot Instances, there is a trade-off between cost and availability. Users need to design their application carefully to be prepared for early termination. Setting a higher maximum bid lowers the probability of early termination, but this may lead to higher cost. It is difficult to examine this trade-off, which involves factors such as price expectation and the cost of tolerating interrupts.

Within the context of the Supercloud, we have developed *Smart Spot Instances*: Instead of making a trade-off between cost and availability, Smart Spot Instances can achieve low cost and high availability at the same time with the support of user-level live migration. User VMs run as nested or *second layer VMs* (*sVMs*), while the Amazon Spot Instances form the *first layer VMs* (*fVMs*). Nested virtualization allows sVMs to be migrated according to a scheduling policy.

Smart Spot Instances can increase availability compared to ordinary Spot Instances. Users set a high maximum bid, which can be as high as the On-Demand prices. The Supercloud monitors the Spot price and live migrates sVMs to other, possibly cheaper fVMs – either just another Spot Instance type, or even Spot Instances in another availability zone, another region, or another cloud. In the worst case, the sVMs can be live migrated to regular On-Demand Instances that are always available, so the user only needs to pay as much as the On-Demand price. When the price approaches the maximum bid in the middle of an instance hour, the Supercloud migrates the sVMs to avoid being terminated.

The challenge of the Smart Spot Instance Scheduler is to place sVMs on fVMs where different types of fVMs will be able to hold different numbers of sVMs. Complicating matters further, fVM prices change from time to time. Because of the significant price difference between fVM types, moving sVMs between types could result in large cost savings. Thus, the contributions of this paper are the following.

- We present a scheduler that optimizes placement of groups of sVMs;
- We show that it addresses the tradeoff between terminating a Spot Instance versus maintaining high availability;
- We demonstrate benefits of using Smart Spot Instances, both by evaluating a system deployment and by exploring the solution space via simulation.

Our previous work (Jia et al. 2015) considered scheduling only a single instance and was entirely synthetic.

2. Challenges and Opportunities

The Smart Spot Instance Scheduler deals with scheduling multiple sVMs on different types of fVMs, either Spot Instances or regular instances. This section discusses challenges in formulating the best scheduling policy and opportunities that may be attained.

2.1 Price Trends in Different Types of Spot Instance

Figure 1b shows the Spot price variation of different instance types between March 25 and March 27, 2015 in the `us-east-1b` availability zone. For each instance type, the graph shows the ratio between the Spot price and the *regular* On-Demand price for that type. While the Spot price of `c3.large` and `c3.xlarge` were relatively stable, instance types `m3.large` and `m3.xlarge` had large price variations. Some spikes in the `m3.large` Spot price even exceeded the regular price. The price for `m3.xlarge` Spot Instances fluctuated between the regular price and the lowest price. As we surveyed other time periods and other availability zones, the price history usually shows a significant difference between instance types.

2.2 Multiple Resource Allocation

When placing sVMs into fVMs, the Smart Spot Instance Scheduler needs to consider the capacity of different resources, including number of vCPUs, memory size, network bandwidth, and disk throughput. In this paper, we will not consider oversubscription, which means that the scheduler will always guarantee the resources requested by the user. To implement this, the scheduler stops placing sVMs into fVMs until sufficient resources free up.

Although this will lead to under-utilization of some resources, this approach guarantees the quality of service requested by the sVMs.

2.3 Different Instance Hour Start Times

Spot Instances are charged at the beginning of each instance hour. If the next instance hour increases in price, then we may need to migrate sVMs away from the Spot Instances before the next instance hour begins. In order to finish live migration before the end of an instance hour, the Smart Spot Instance Scheduler needs to leave a large enough *margin time* during which the Supercloud can allocate a new Spot Instance in a cheaper location and finish the migration before the start of the next instance hour. During this margin time, users pay for both Spot Instances.

When making the decision at the end of the instance hour, some of the instances might continue to run while some instances are cleared and the sVMs on them are migrated to other types of instances. The new fVMs start the next instance hour earlier than the ones that continue to run. A challenge is that a placement decision needs to be made slightly before any of the fVMs finish their instance hour, but the ending of fVM instance hours may be out of sync.

Another challenge is that the placement decision is purely reactive: The scheduler considers only current prices of instance types in its placement decision. If, on the other hand, we could predict prices, then the placement decision could be more proactive. However, studies (Mazzucco and Dumas 2011) and our own observations (Figure 1) demonstrate that predicting price remains elusive. As a result, the Smart Spot Instance Scheduler only considers reactive placement decisions based on current instance type prices in this paper.

2.4 Migration across Availability Zones and Providers

Each availability zone has its own Spot Market, hence the same type of Spot Instance may be charged differently in different zones at the same time. Our techniques make it possible to migrate sVMs to another availability zone, or even to a different cloud provider. But data transfer across availability zones costs \$0.02/GB (\$0.01/GB for both ingress and egress traffic), and across providers is even more expensive. When live migrating sVMs, we need to migrate the memory and the dirty pages generated during the migration time, so migrating even a single sVM across availability zones incurs significant cost—at least several cents—close to the price of a regular On-Demand Instance and usually much more than the price of a Spot Instance.

While the migration cost is a one-time cost, and migrating to a cheaper availability zone might eventually be worth the investment of migration, we cannot easily predict that the availability zone will continue being cheap. In order to make migration across availability zones beneficial in terms of cost, the best way is to reduce the data transfer during the migration. A lot of techniques can be used, such as memory de-duplication and self-ballooning. However, for this paper, the Smart Spot Instance Scheduler will only migrate sVMs within the same availability zone.

2.5 Reliability Requirements

Amazon reserves the right to terminate Spot Instances at any time. Although Smart Spot Instances can improve reliability compared to normal Spot Instances, their reliability is still not as good as that of regular instances. One possible solution is to replicate sVMs in different availability zones. However, for some applications the networking cost of synchronizing replicas could be high. A compromise between cost and reliability is to run replicas in different types of fVMs within the same availability zone, as networking cost would be free. Different fVM types have different (price) markets, so would be less likely to be terminated at the same time.

3. Scheduling Smart Spot Instances

We designed and implemented a centralized scheduler to select the types of fVMs and place sVMs on them. The scheduler maintains the current placement of sVMs in the Supercloud. It monitors the Spot price and decides the types of fVMs and placement of sVMs when fVMs approach the end of their instance hour. Specifically, the scheduler determines the best combination of fVM types, allocates new fVMs if need be, deploys the Supercloud library environment on the newly started fVMs, clears fVMs to be terminated by migrating its sVMs to other fVMs, and terminates the former fVMs after migration finishes. We consider only one type of sVMs, that is, sVMs all have the same number of vCPUs, memory size, network bandwidth, and disk access speed.

Should the scheduler becomes a bottleneck, we could partition the resources and use multiple schedulers at reduced optimality.

3.1 Scheduler

The margin time (Section 2.3), denoted as t_M, is used to allocate new instances and migrating the sVMs before the expiration of the current instance hour. The margin time is configurable and can be tuned based on VM size and available bandwidth between the two fVMs. Since the start times of the fVMs are not synchronized, we divide each hour into slots that have a length equal to the margin time and run the scheduler at the beginning of each slot. The scheduler then considers the overall placement of sVMs on all the fVMs that are going to finish their instance hour during the current slot and use the current slot to finish migration.

The scheduler obtains the capacity and price of regular and Spot fVM types, the collection of fVMs that are finishing within the margin time (*finishing fVMs*), and the current sVMs on them (number and sizes). The goal of the placement algorithm is to find the most cost-effective way to place the sVMs on the finishing fVMs. The scheduler could either continue to work with the current set of finishing fVMs or allocate new fVM instance types while allowing some (or all) finishing fVMs to terminate. The scheduler migrates all the sVMs away before terminating the fVMs.

Specifically, let n be the number of finishing fVMs, and let m be the number of sVMs running on them. The scheduler determines the set of fVMs to be used in the next instance hour. However, since we cannot predict future prices, the algorithm can only do local optimization with all information available at the time of invocation. The local optimization goal is to minimize the total cost C for running the m sVMs in the next instance hour.

The calculation of cost C for any algorithm needs to consider the cost of keeping a finishing fVM and keeping all of the sVMs associated with it (No Migration) versus terminating a fVM and migrating associated sVMs to a new fVM (Migration). See Figure 2 for an illustration of this scheduling decision. In particular, suppose we want to terminate a finishing fVM and migrate all sVMs on it to another fVM with price p_{new} (Migration case in Figure 2). The cost of running these sVMs in the next instance hour will be p_{new} (i.e. the cost of the new fVM, which will host the sVMs). However, suppose we keep the finishing fVM (No Migration case in Figure 2). We assume its price for the next instance hour would be the same as the observed price at the beginning of the margin time when the scheduler was invoked, we call this price p_{curr}. The cost of the next instance hour will be reduced by the margin time t_M since it was already paid for during the previous instance hour. Thus, the cost of the next instance hour will be $p_{curr}(1-t_M/60)$ (i.e. the price of the finishing fVM multiplied by the next instance hour minus the margin time divided by an hour). As a result, the scheduler uses p_{new} versus $p_{curr}(1-t_M/60)$ to decide whether or not to allocate new fVMs and migrate their associated sVMs.

In the following subsections, we describe two scheduling algorithms: Greedy and dynamic programming, in Sections 3.2 and 3.3, respectively. The latter achieves a local optimal placement and cost.

3.2 Greedy Algorithm

The greedy algorithm works as follows. For the m sVMs on finishing fVMs, consider all fVM types, one type at a time, and for each fVM type, compute the total cost to run all m sVMs. Then, select the fVM type with the minimum cost.

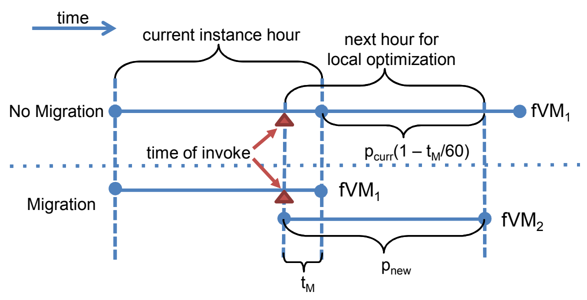


Figure 2. Timeline of Scheduling Decisions

For an fVM type j with price p_j and capacity[2] c_j, the cost of running all m sVMs is $p_j\lceil m/c_j\rceil$ where $\lceil m/c_j\rceil$ is the number of fVM instances necessary to hold m sVMs. If the fVM type is the current type of fVM, we use the price $p_j(1-t_M/60)$ as described in Section 3.1. The greedy algorithm selects the fVM type with the minimum cost.

There are two advantages for the greedy algorithm. First, it is simple. Only one fVM type is used during a scheduling slot, which simplifies the algorithm. Second, if m is a multiple of the capacity c_j for fVM type j that the greedy algorithm selects, the placement returned by the greedy algorithm would be optimal. Otherwise, the resources for $m \mod c_j$ sVMs will be wasted. If m is large, $m \mod c_j$ will be a small fraction of m and thus the greedy algorithm approximates the optimal solution.

3.3 Dynamic Programming Algorithm

In this section, we use a dynamic programming algorithm to find the optimal placement for the m sVMs. Dynamic programming algorithms solve problems that exhibit the property of *optimal substructure*.

We first consider a simpler problem that placing m sVMs onto k different fVM instance types to achieve the minimum cost and do not consider the overlapping margin time cost. We want to get the solution of the problem from the cheapest placement of x sVMs, with x ranging from 0 to $m-1$. Note that the problem of minimizing the cost of placing x sVMs can be solved by examining the cost of placing d sVMs into a *single* fVM with instance type j and the sum of the minimum cost of placing $x-d$ sVMs, for all possible js and ds. So the problem can be solved using the following dynamic programming equation, with $f(x)$ representing the minimum cost to run x sVMs.

$$f(x) = \underset{0\le j<k,\ 0<d\le c_j}{\arg\min} \quad (f(x-d)+p_j)$$

In this equation, c_j stands for the capacity of the j^{th} fVM type, and p_j is its current price. We treat the On-Demand Instance as a special kind of Spot Instance whose price remains unchanged.

We compute $f(x)$ with x ranging from 0 to m. $f(0)=0$ because it is free to run 0 sVMs. $f(m)$ will be the minimum cost to run the m sVMs on the finishing fVMs. The choice of the fVM types can be determined by storing the choice js and ds that minimizes $f(x)$ with x ranging from 0 to m.

In order to incorporate the margin time overhead, we need to modify the original dynamic programming algorithm. If the scheduler chooses to keep the finishing fVMs for another hour, the start time will be t_M minutes later. Now our price function f takes two parameters, x and i. i indicates that the function has considered

[2] Capacity c_j would correspond to the most constrained resource for an fVM of type j; e.g. minimum between CPUs, memory and network bandwidth, relative to the resources required by an sVM.

the first i finishing fVMs, which can either be chosen to continue or terminate, determined by the cost of either case. Solving $f(x,i)$ consists of two subproblems. The first subproblem is placing d sVMs into a single instance of type j and getting the minimum cost of running $x-d$ sVMs with first i finishing VMs already considered, for all possible js and ds. The second subproblem is keeping the i^{th} finishing VM and adding up the cost of using the i^{th} VMs to run d sVMs and the minimum cost of running $x-d$ sVMs, for all possible ds.

The dynamic programming equation contains two argmins, which solve the two subproblems respectively, as follows:

$$f(x,i) = \min\Big(\operatorname*{arg\,min}_{0\le j<k,\ 0<d\le c_j} (f(x-d,i)+p_j),$$
$$\operatorname*{arg\,min}_{0<d\le c_{type[i]}} (f(x-d,i-1)+p_{type[i]}(1-t_M/60))\Big)$$

$type[i]$ is used to represent the type index of the i^{th} finishing fVM. The minimum cost to run the m sVMs on n finishing fVMs will be $f(m,n)$, which can be computed from $f(0,i), i \in \{0,1,..,n\}$ with $f(0,i)=0$.

The complexity of the running time is $O(m\cdot n\cdot k\cdot c)$, where c is the largest capacity of all fVMs. k will be small, since there are only a few instance types. c is also small because even the largest fVM can hold a few sVMs. When scheduling less than tens of thousands of VMs, this procedure can be reasonably fast. Although it is not as scalable as the greedy algorithm, it is likely to achieve lower cost.

3.4 Greedy VM Replication Algorithm

Section 2.5 described how Supercloud users can use sVMs as replicas and ask the scheduler to place the replicas into separate fVM types to improve availability (we assume that fVM types fail and terminate instances differently since they follow different markets). In particular, the greedy VM replication algorithm works the same as the greedy algorithm described in Section 3.2, except that it is invoked with all replica groups of the same size with the constraint that it has to use as many different fVM types as the replica group size. In particular, if a replica group size is r, instead of choosing the cheapest fVM type, the greedy VM replication algorithm will choose the r cheapest fVM types. Finally, it is invoked separately for replica groups of different sizes.

4. Evaluation

The goal of the evaluation is to see if using user-level migration in leveraging the Spot Market leads to substantial price savings while maintaining availability. We evaluate both the dynamic programming and the greedy scheduling algorithms and compare them with other approaches. In order to have more control and reproducibility we first use simulation to evaluate the algorithms, and then run some experiments using TPC-W on a Supercloud deployment.

4.1 Comparison of Approaches

We used simulation to compare different algorithms for scheduling Smart Spot Instances. The cost savings depended on the price history. For our experiments, we used two 10-day price history traces obtained from Amazon. The margin time t_M was set to 5 minutes in all our evaluations. We compared three classes of scheduling strategies:

- *Dynamic*: our dynamic programming (DP) and greedy (GR) algorithms dynamically place and migrate sVMs;
- *Static Spot*: this strategy sticks with the initial placement of the sVMs.The initial placement can either be chosen from a single Spot Instance type, or the best placement on a combination of Spot Instance types, according to the Spot prices at the beginning of the period (called *Combo*).
- *Static Regular*: this strategy uses a single type of regular On-Demand Instance throughout.

The first ten-day price history starts at 0:00am 3/25/2015 UTC in Amazon's `us-west-2a` availability zone. Spot Instance types were chosen from `c3.large`, `c3.xlarge` and `c3.2xlarge` which have 2, 4 and 8 vCPUs resp. The memory sizes were 3.75, 7.5 and 15GB resp. The sVMs had a size of 1vCPU and 1GB memory. We left 1vCPU and 3GB memory for the second layer Dom0 and OpenStack VM, so the maximum number of sVMs that can be hosted on the different types of Spot Instances were 1, 3, and 7 resp. Although a `t2.micro` instance provides 1vCPU and 1GB memory, the CPU frequency, disk IOPS, and network bandwidth are much lower than for a `c3` instance, so we do not compare with running the application directly on `t2.micro` instances.

Figure 3a and 3b show the cost of running 10 and 100 sVMs during this period. Dynamic placement achieved more than 2x cost savings compared to other placement strategies. With only 10 sVMs, the greedy algorithm ended up costing about 16% more than dynamic programming, but with 100 sVMs, the cost is less than 10%. The Combo strategy cannot protect against price changes in the Spot Market, and did only moderately better than sticking to a single Spot Instance type, and in fact could perform worse if the initial placement ends up costing more in the long run than some instance type. The dynamic approaches ended up being approximately 4x cheaper than placing the sVMs onto regular On-Demand Instances.

We also used a price history from the `us-east-1b` availability zone starting at 0:00am 4/1/2015 UTC. Instead of `c3` instances, we used `m3.large`, `m3.xlarge` and `m3.2xlarge`, which have the same number of vCPUs but twice the memory compared to `c3` instances. They also host 1, 3, and 7 sVMs resp. with a size of 1vCPU and 3GB memory. Compared to the previous price history, the Spot Market prices were significantly more stable. In particular, the price of `m3.2xlarge` stayed low during the entire period.

Figure 3c and 3d show that running sVMs in Spot Instances were at least 5x cheaper than regular instances, even without migration. Migration achieved moderate cost savings compared to Combo or any of the single instance types.

4.2 VM Replication

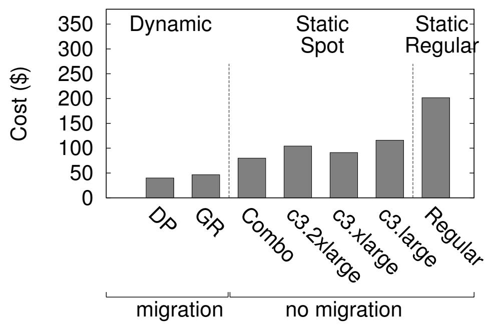
We used simulation to evaluate the cost of a 2-replication group using the previous two price histories with 100 sVMs, for a total of 200 sVMs, and used the greedy algorithm to solve the replication constraints. Figure 4 shows the results. Because we replicate VMs to achieve better availability in the face of Spot Instance termination, we did not replicate to regular instances.

The greedy algorithm achieved lower cost than all possible pairs of instances types to hold the replicas. Even in the case when the prices were relatively stable, the algorithm achieved more than 15% savings compared to the best static choice of instance types.

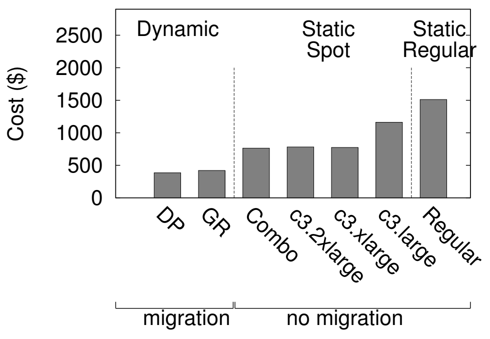
4.3 TPC-W Benchmark

We evaluated the dynamic programming algorithm using the TPC-W web benchmark. 10 sVMs ran as 10 different TPC-W servers. Each server served requests from different sets of TPC-W clients. The TPC-W clients were hosted by `m3.large` instances, each of which hosted two client sets and joined the Supercloud VPN to access the TPC-W servers using private IP addresses. Since migration in the Supercloud does not change the server's private IP address, we did not need to reconfigure clients after migration. The setup and history were the same as the second price history shown in Section 4.1, except that we only ran the experiment for the first day.

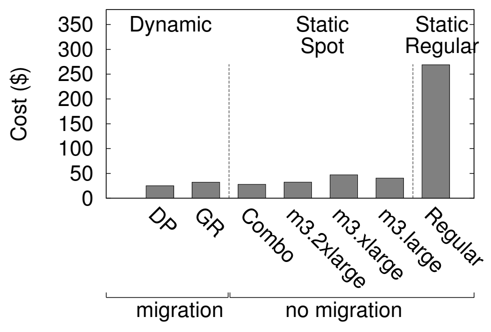
Figure 5a shows the accumulative prices of running all 10 TPC-W servers as sVMs with different placement schemes. The Smart



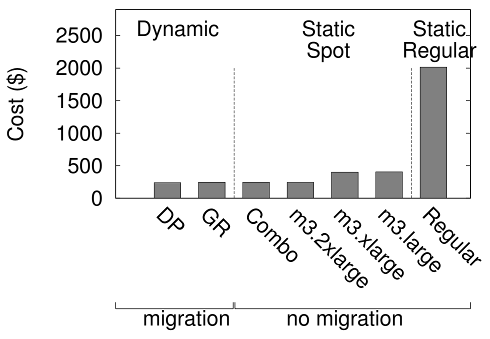
(a) 10 sVMs in us-west-2a



(b) 100 sVMs in us-west-2a



(c) 10 sVMs in us-east-1b



(d) 100 sVMs in us-east-1b

Figure 3. Cost Savings in Simulation Experiments.

Spot Instance using the DP algorithm triggered migrations six times (denoted as the dashed lines) during the experiment and maintained the lowest cost over the entire experiment. In contrast, the static deployments, either on single instance types or combining multiple types, suffered from price spikes. These spikes typically lasted one to two hours. Figure 5b shows that the performance of the Smart Spot Instance remained high throughout the experiment.

5. Discussion

The evaluation in Section 4 is based on experiments with the Amazon Spot Market, but we believe the Supercloud and Smart Spot Instances can also be used with Google Preemptable Instances to improve availability and save cost. Below we discuss how Google Preemptable Instances differ from Amazon Spot Instances and the required changes for the Smart Sport Instance scheduler to work with Google Preemptable Instances.

Google Preemptable Instances differ from Amazon Spot Instances in several ways.

- Prices of Preemptable Instances are fixed–in particular, prices do not change based on time or location;
- The maximum length of a Preemptable Instance is 24 hours–after 24 hours, an instance is terminated;
- the grace period before a Preemptable Instance is preempted (terminated) is only 30 seconds.

To adapt to the specifics of Google Preemptable Instances, the Smart Spot Instance scheduler will have to be changed in the following ways.

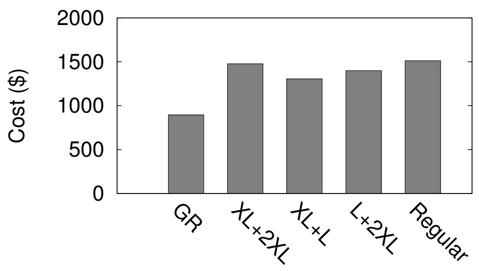
- Because of the fixed price of Google Preemptable Instances, the scheduler does not need to monitor the price and migrate sVMs accordingly. However, the scheduler would still use either the Dynamic Programming or Greedy algorithm to place sVMs onto the fVMs during scheduling decisions;
- The period of placement decisions increases to once every 24 hours. The scheduler still needs to keep track of the running time of each fVM. When the running time of the fVM approaches 24 hours, the scheduler needs to compute a new placement of the sVMs on the fVM.
- Due to the 30 second preemption (termination) grace period, Smart Spot Instances can only support fairly small fVM instances that can be live migrated within 30 seconds. (For future work, we are investigating containers which can be migration much faster.)

The Supercloud can also span different cloud providers and is able to schedule sVMs across Google and Amazon at the same time. When choosing fVM providers and instance types, the scheduler can treat Google Preemptable Instances as different types of instances with fixed prices. The scheduler would need to consider that migration cost between providers is even more expensive than migration between availability zones, as discussed in Section 2.4. The scheduler also needs to adapt to the different cloud APIs.

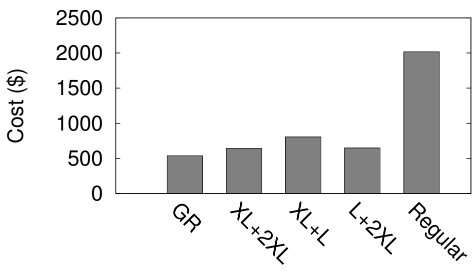
6. Related Work

VM live migration has been widely used for resource consolidation and workload burst handling (Wood et al. 2007). However, existing techniques mainly focus on reducing the number of physical hosts being used. Smart Spot Instances need to additionally consider unique challenges that stem from monetary costs in the cloud and the pricing model and dynamic market of the Amazon Spot Market.

There have been studies on better utilizing Amazon Spot Instances to increase performance and lower cost. (Chohan et al. 2010) propose a Markov-based job scheduler for MapReduce applications that adds Spot Instances as accelerators when the envi-

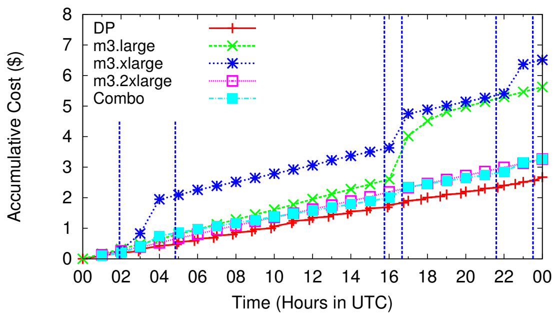


(a) us-west-2a

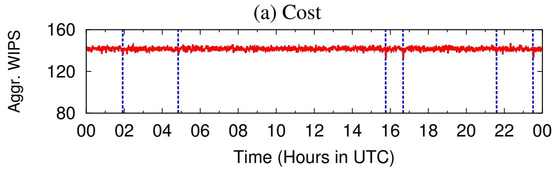


(b) us-east-1b

Figure 4. Cost of Replicating 100 sVMs. m3.large, m3.xlarge and m3.2xlarge are denoted as L, XL, and 2XL.



(a) Cost



(b) Performance

Figure 5. Cost and performance of Smart Spot Instances. Migrations in DP are indicated as vertical dashed lines.

ronment (i.e., the Spot Instance price) is favorable. (Yi et al. 2010) study the mechanisms and tools that deal with the cost-reliability trade-offs in the Spot market. (Mazzucco and Dumas 2011) analyze the strategies required to achieve high-availability based on price prediction and a cost-reward model. Studies show that HPC applications can also benefit from Spot Instances (Taifi 2013). Compared to these works, a major advantage of Smart Spot Instances is the ability to migrate running applications. Thus, applications are not locked in to any particular Spot Instance, and it is much easier for users to exploit price differences. SpotCheck (Sharma et al. 2015) used checkpointing and migration to improve reliability of spot instances while Smart Spot Instances focus on reducing cost.

A Cloud Service Brokerage acts as a front-end for multiple cloud providers assisting users with selecting the right cloud or clouds for running their applications (Barker et al. 2015). The Supercloud can be thought of as such a brokerage. To the best of our knowledge, the Supercloud is the only brokerage with specific support for managing VMs in a Spot Market.

7. Conclusion

This paper described some initial investigation of scheduling a group of virtual machines in the Amazon EC2 Spot market, leveraging live user-level VM migration to overcome the inability to predict future prices. Much work still needs to be considered. Currently, we require that all VMs are the same size and have not yet investigated the effect on storage and networking. We also hope to find use cases where multiple cloud providers can be exploited for further price savings.

Availability

The Supercloud project website is located at http://supercloud.cs.cornell.edu, and the code is publicly available.

Acknowledgments

The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, by the AFOSR MURI Science of Cyber Security: Modeling, Composition, and Measurement as AFOSR grant FA9550-11-1-0137, by NSF grants CNS-1601879, 0430161, 0964409, 1040689, 1047540, 1053757, 1151268, 1422544, 1518779, 1561209, and CCF-0424422 (TRUST), by ONR grants N00014-01-1-0968 and N00014-09-1-0652, by NIST grant 60NANB15D327, by DARPA grants FA8750-10-2-0238, FA8750-11-2-0256, and D11AP00266, by MDCN/iAd grant 54083, and by grants from Microsoft Corporation, Infosys, Google, Facebook Inc., and Amazon.com.

References

A. Barker, B. Varghese, and L. Thai. Cloud services brokerage: A survey and research roadmap. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 1029–1032. IEEE, 2015.

N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using spot instances for MapReduce workflows. In *HotCloud'10*, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Supercloud: Opportunities and challenges. *SIGOPS Oper. Syst. Rev.*, 49(1): 137–141, Jan. 2015. ISSN 0163-5980.

M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *HPCC 2011*, pages 296–303, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4538-7.

P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative IaaS cloud on the spot market. EuroSys '15, pages 16:1–16:15, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5.

M. Taifi. Banking on decoupling: Budget-driven sustainability for HPC applications on auction-based clouds. *SIGOPS Oper. Syst. Rev.*, 47(2): 41–50, July 2013. ISSN 0163-5980.

T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI'07*, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In *IEEE CLOUD 2010*, pages 236–243, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4130-3.