Reliable Approach to Sell the Spare Capacity in the Cloud

Wesam Dawoud
Hasso Plattner Institute
Potsdam University
Potsdam, Germany
wesam.dawoud@hpi.uni-potsdam.de

Ibrahim Takouna
Hasso Plattner Institute
Potsdam University
Potsdam, Germany
ibrahim.takouna@hpi.uni-potsdam.de

Christoph Meinel
Hasso Plattner Institute
Potsdam University
Potsdam, Germany
christoph.meinel@hpi.uni-potsdam.de

***Abstract*—Traditionally, Infrastructure as a Service (IaaS) providers deliver their services as *Reserved* or *On-Demand* instances. Spot Instances (SIs) is a complementary service that allows customers to bid on the free capacity in the provider data centers. Therefore, the decrease in the free capacity may result in terminating instances abruptly. To ensure fair trading, the provider does not charge customers for the interrupted partial hours. However, our experiments show that uncharged time could rise up to 30% of the instance total run time, which means a reduction in the provider's profit. In this paper, we propose an Elastic Spot Instances (ESIs) approach, where instead of abruptly terminating the SI, the provider scales down their capacity proportionally to the increase in the price. Our approach delegates the task of interrupting the instances into the customers, but at the same time keeps the control in the provider side to isolate SIs' impact on the other services. We validate our approach along different periods of SIs history traces.**

Keywords- IaaS; Spot instances; Dynamic scalability.

I. INTRODUCTION

Amazon is the first cloud provider to come up with SIs purchasing system to sell the spare capacity after fulfilling the requests for *Reserved* and *On-Demand* instances. The price of SIs changes dynamically according to free capacity and actual demand. The requests for new SI with bid price higher than or equal the current spot price will be served. On the other hand, if the current prices exceeded the user bid, provider will terminate out-of-bid instances abruptly. SIs reduce the prices from 38% to 44% of the *On-Demand* prices [1]. However, SIs customers are supposed to modify their applications to manage the abrupt termination of SIs.

To manage SIs termination, customers can implement fault tolerant architectures such as MapReduce [2], Grid, Queue-Based [3], and Checkpointing [4][5][6]. The first three architectures typically run two types of nodes (master and worker). One of the master nodes tasks is to manage the failure of worker nodes. The best practice is to run master nodes on *On-Demand* or *Reserved* instances and run worker nodes on SIs to benefit from price reduction. However, these architectures imply major modification to customers' applications. On the other hand, checkpointing is a simple traditional fault tolerant technique. It keeps application execution progress by storing the current state (i.e., snapshot) of the running instance into a persistent storage. Nevertheless, bad checkpointing strategies could impact the performance drastically [7]. For instance, frequent checkpointing results in a high cumulative overhead (i.e., computation is paused at checkpointing time). On the other hand, infrequent checkpointing results in a high overhead caused by the high recovery time (i.e., much computation should be repeated again).

The main goal of this paper is to reduce the checkpointing overhead in SIs environment. This is motivated by the following facts: First, checkpointing is a simple fault tolerant technique that does not require major modifications to customers' applications. Second, checkpointing could be integrated to the other fault tolerant architectures to increase their reliability. Finally, and most importantly, if customers can have checkpoints exactly before terminating VMs instances (i.e., Optimal Checkpointing), then there is no need for the concept of unpaid partial running hours, which on consequently increases the provider profit.

In the next section, we study Amazon EC2 SIs implementation. In Section III, we discuss our proposed ESIs approach: the algorithm, the advantages, and the potential technical challenges. In Section IV we compare our proposed approach performance with current implementation of SIs using price history traces. In Section VI, we present related work done to improve the trade-off between price, reliability, and total run time of applications on SIs. Finally, in Section VII, we conclude and represent our future work.

II. AMAZON EC2 SIS

In this section, we give an overview to Amazon EC2 SIs because it is the first provider who offers SIs purchasing system. The purpose of this section is to determine SIs characteristics that we should consider in our approach.

A. Infrastructure

Amazon EC2 infrastructure [8] is distributed into regions (e.g., US East "Northern Virginia", US West "Northern California", etc.). To prevent failure propagation, each Region is separated into many availability zones. This infrastructure mainly delivers *Reserved* and *On-Demand* instances. The spare capacity is sold as SIs. The SI, as well as the *Reserved* and *On-Demand* instance, can be one of many types depending on resources capacity (e.g., High-CPU Medium

 ISBN: 978-1-61208-216-5

Instance "c1.medium", High-Memory Extra Large Instance "m2.xlarge", etc.).

SI's price is determined by the type, the region, and the operating system. Unlike Zhang et al.'s [9] assumption, in our approach we assume that a physical machine, at the provider side, hosts only instances of the same type and operating system. We support our assumption by observing CPU specifications of each EC2 instance type.

B. Is it a market-driven auction?

Amazon describes SIs purchasing system as a market-driven auction [3]. For example, if the provider has N free resources and it received K bids on the resources, then the provider accepts only the highest N bids, where K is greater than N. The price will be the lowest bid value of the winning subset of the bids. However, by analyzing history traces of SIs' price, Javadi et al. [1] showed sharp changes in the *inter-price* time (i.e., time between price changes) occurred on specific dates at different regions. Javadi et al. conclude that it is artificial (i.e., done by Amazon and not driven by customers demand).

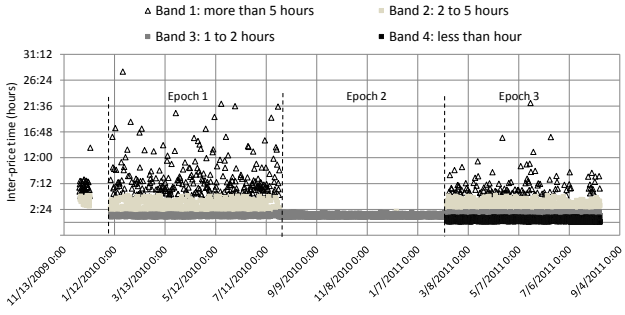

Fig. 1. (US-East), High-CPU Medium Instance's *inter-price* time

Ben-Yehuda et al. [10] went further by showing that the prices also are determined artificially by a random reserve price algorithm and do not represent real customers bids around 98% of the time. They expected that the aim of this random reserved price is to prevent customers from: 1-being complacent and force them to bid higher. 2-inferring the provider's real capacity. Therefore, in case of low number of bids on specific instances type (i.e., K is lower than N), the provider either accepts the lowest bid as the current price or generates a higher value (i.e., pretends less resources [9]) to sell the resources with a higher price. This behavior raises a question about the efficiency of approaches that model the SIs prices. However, we attribute the direct control of the provider on the price to the lack of demand for some instances' type. Nevertheless, regardless of the aim of this random reserved price, our approach does not disclose any hidden information about provider's capacity while it proposes changes to purchasing system rather than to pricing system of SIs.

III. Elastic Spot Instances (ESIs)

We propose ESIs approach to increase the efficiency of selling the free capacity of the IaaS provider. It assumes modifying the SIs purchasing system to increase its reliability without influencing the other hosted services. Fortunately, these modifications do not imply major modification to the current providers' infrastructure while many IaaS providers already use virtualization technologies that can easily accommodate our approach.

Current SIs implementation reacts with the increase demand on *On-Demand* and *Reserved* instances by increasing the SIs price. As a result, out-of-bid SIs are evicted to free more capacity for the complementary services. From the customer point of view, this reduces the SIs reliability. From the provider side, the abrupt interruption of the SIs results in partial unpaid hours (i.e., in some cases, provider will not be paid up to 30% of an instance total run time). Moreover, if the users managed to delay the SI termination, as discussed by [11] and [5], this also increases the probability of unpaid running time.

Implementing our approach requires the following modifications to current SIs purchasing algorithm: First, provider should determine min and max price for each instance type. Second, instead of terminating out-of-bid SI the provider scales down instance's capacity to a value proportional to the increase in the price. Third, running instances can be charged per second because VM instance termination is delegated to the user. According to these modifications, the capacity of ESI can be calculated using Algorithm 1.

Algorithm 1 ESIs's purchasing algorithm

Input: max_price, min_price, current_price, min_cap, and user_bid
Output: VM_capacity
// Calculate the scaling step size
$scale\_step \leftarrow 100/(1000 * (max\_price - min\_price) + 1)$
// Calculate next capacity of VM
if $user\_bid \geq current\_price$ **then**
  $VM\_capacity \leftarrow 100$
else
  if $user\_bid < current\_price$ **then**
    $VM\_capacity \leftarrow 100 - scale\_step * 1000 * (current\_price - user\_bid)$
  end if
  //To prevent VM from starving
  if $VM\_capacity < min\_cap$ **then**
    $VM\_capacity \leftarrow min\_cap$
  end if
end if

According to [1], the price history of most SIs types, except for some types in US-East data center, could be modeled as a Mixture of Gaussian distributions with three or four components with a high fit. This gives the impression that Amazon already has soft minimum and maximum price thresholds for each spot instances type. Moreover, to prevent negative and very low capacities of VM instances, we propose having a minimum capacity of the VM resources, as seen in Algorithm's 1 input. In Section V, we discuss calculating this value considering the free capacity at the physical host.

 ISBN: 978-1-61208-216-5

Algorithm 1 shows that the provider will not have the control to terminate the SIs. At first glance, it seems that customers will be complacent and can simply use a very low bid strategy to have a continued run with a low price. However, if we take the example of "US-West, Linux, High-CPU Medium" instance, the probability density function shows that 99.8% of the prices fall between 0.076 and 0.084. Therefore, scale_step value in Algorithm 1 is calculated as $100/(1000 * (0.084 - 0.076) + 1) = 11.11$, which means that whenever the Spot Price surpasses user bid with 0.001, the capacity of the instances scales down to $(100 - 11.11) \approx 89$. If a user submitted a low bid, for example 0.077, the user will be charged 0.077 per hour for a full capacity instance (i.e., 100%). However, when the market price jump to 0.081, the instance capacity will be scaled down to $100 - 11.11 * (0.081 - 0.077) \approx 56\%$. In spite of the fact that the instance is charged 0.077 per hour, the price is almost doubled according to the low allocated capacity. By this concept, at the case of the overloading, instead of terminating the instances by the provider, the high ratio of price to capacity will push the customers to mange terminating SIs for the optimal price. In Section IV-C, we will discuss the bidding strategies on the light of the proposed modifications to SIs purchasing system.

A. Technical Challenges

Our approach depends on the virtualization technologies' ability to scale the virtualized resource dynamically. However, isolation is a prerequisite for virtualized resources' scalability. It is a demanding problem attracts many researchers [12], [13], [14]. In this section, we discuss isolation and scalability of the following resources: CPU, I/O, and Memory.

CPU isolation is the scheduler's responsibility. Each scheduler has policy that controls the assigned capacity and CPU's time for each virtual CPU (vCPU). Schedulers allow users to change the vCPU's configuration dynamically. However, each scheduler has its characteristics that make it suitable for some environments more than others. For instance, Xen [15] has three schedulers: Borrowed Virtual Time (BVT), Simple Earliest Deadline First (SEDF), and the Credit scheduler [16]. Among these schedulers, only SEDF and Credit scheduler have a non work-conserving mode, which enable the scheduler to cap the capacity of the CPU to specific value (e.g., 50% of the CPU capacity). In spite of the fact that SEDF shows less CPU allocation errors compared to Credit scheduler [16], the global fairness of Credit scheduler makes it the best candidate for our approach.

As in the case of CPU isolation, I/O isolation is also of schedulers' responsibility. However, current implementation of the hypervisors shows that an I/O-intensive VM can influence the performance of other VMs. For instance, in Xen [15], I/O device follows a split-driver model. Therefore, only an Isolated Device Domain (IDD) has access to the hardware using native device drivers. Cherkasova [16] and Gupta et al. [17] demonstrated that the I/O model of Xen complicates CPU allocation and accounting since an IDD processes I/O on behalf of guest VMs. To enhance the accounting mechanism, [17] proposed SEDF-DC. It accounts the CPU usage of an IDD into corresponding guest domains that trigger I/O operations. However, SEDF-DC is still a prototype and it is not implemented to the deployed version of Xen. We leave integrating SEDF-DC to our approach to our extended work.

At initialization time of VMs, the hypervisor allocates an isolated virtual memory for each VM. Memory isolation makes the VM unaware of other VMs' or hypervisor memory demand. A Ballooning technique is developed to enable passing memory pages back and forth between hypervisor and hosted VMs. However, it requires the cooperation of the VM's operating system. Therefore, VM's operating system should be plugged with balloon driver to enable the communication between the VM's operating system and the hypervisor. In case that the hypervisor decide to reduce the VM's memory size (i.e., reclaim pages from VM and inflate the balloon [18]), it determines the target balloon size. If the VM's operating system has plenty of free physical memory, inflating the balloon will be done by just pinning free memory pages (i.e., prevent access to these pages). However, if the VM's is already under memory pressure, the operating system should decide about the memory pages that should be paged out to the virtual swap device[18]. In spite of the fact that paging impacts the VMs performance, Ballooning technique shows better performance compared to Hypervisor Swapping reclamation technique [18]. The lack of the knowledge about the pages contents, in case of Hypervisor Swapping, may result in paging VM's kernel, which has a significant impact on the VM performance. In this paper, we used the balloon driver implemented by Xen to scale down the memory of VMs with the price increase. However, the reality of initiated CPU-intensive workload, in our experiments, did not examine the memory scalability performance. Therefore, we leave examining our approach's performance against different kinds of workload to our future work.

IV. Evaluation

To validate our approach, we carried out two sets of experiments on the physical hardware and using simulation. The first set of experiments, carried out on our Xen test bed, focuses on modeling the virtual machine against CPU-intensive workload with different values of CPU capacity. The other set of experiments carried out by feeding our simulator with the extracted model to simulate running a job of 168 hours (one week). We chose this length of job according to [1] observation that the Spot Price follow specific patterns during the weekdays. Moreover, long run jobs gave us consistence results compared with short jobs. The job run is simulated on a SI and on ESI using SIs' price history traces that are gathered by [19].

A. VM model for CPU-intensive workload

To extract the VM instance model, we ran a VM with two cores on Xen 4.1 hypervisor. The physical server has 2.8 GHz Intel Quad Core i7 Processor and 8GB of physical memory. The workload is CPU-intensive workload generated

 ISBN: 978-1-61208-216-5

by EP Embarrassing Parallel, which is one of NAS Parallel Benchmarks (NPB) [20]. The benchmark generates independent Gaussian random varieties using the Marsaglia polar method. The throughput is measured by Million Operations Per second (MOPs).

At the beginning, the VM instance runs with its full capacity (i.e., 100%). As seen in Fig. 2, the throughput is 37.92 MOPs and the execution time is 56.6 seconds. The same workload is run many times but for different capacities of the VM's CPU. In our experiment, we use Xen *Credit Scheduler* as an actuator for setting the CPU capacity limit of the VM. The *Credit Scheduler* has a non work-conserving mode, which prevents an overloaded VM from consuming the whole CPU capacity of the host and consequently degrading the other VMs performance. For each CPU capacity, we recorded both the MOPs number and the total execution times.

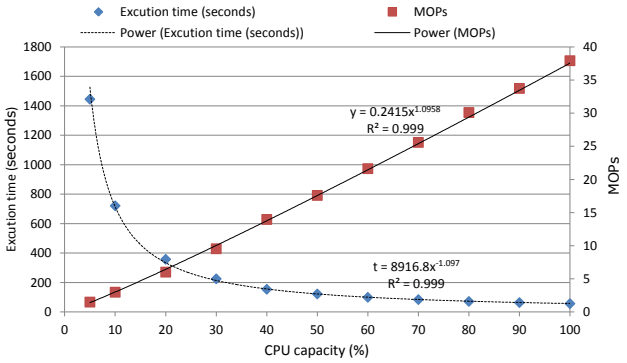

Fig. 2. VM's model against CPU-intensive workload

As seen in Fig. 2, the instance's throughput changes linearly with the virtual CPU (vCPU) capacity according to the following equation:

$$0.2415 * x^{1.0958} \quad (1)$$

where x is the vCPU capacity. However, as the capacity of the VM's vCPU decreases, the execution time increases. At very low capacities (i.e., less than 20%) the execution time increases rapidly. Therefore, in Section IV-C, we explain the bidding strategies that avoid inefficient run for VMs instances.

B. SI simulation

In this section, we simulate running a job of 168 hours on a SI. We chose High-CPU Medium Instance (c1.medium) type while it is an instance type offered to deliver high CPU computation power. Moreover, we selected US-East Region specifically because it shows different values for *inter-price* bands. We would like to study the influence of these bands on the provider profit (i.e., the percentage of unpaid computation hours), as well as on the SI's performance.

In our simulation, we use optimal checkpointing strategy (i.e., checkpointing exactly before the instance termination). During checkpointing time, the computation in VM is paused [11]. Moreover, restoring a VM mounts additional overhead to the checkpointing technique. Sotomayor et al. [21] provide a model to estimate the suspension and restoration time of a VM. The model depends on the number of the co-located VMs and the storage location (i.e., local or remote). However, we cannot predict the number of co-located VMs at public cloud providers. Therefore, we depend on measuring the time required for having a snapshot of c1.medium instance type. Measurements are done 10 times on different time slots of the day at US-East Region. The measured value was always less than a minute. On the other hand, measuring restoration time was ambiguous. Even with very high bids, the measured time between submitting a request and running a SI, from a snapshot, was measured to be 7 to 10 minutes. It is clear that, the bidding algorithm impacts restoring a SI even for very high bids. Therefore, in our simulation, we use the suspension and restoration time which have been estimated by [21] for two VMs to a remote storage. The values are 120 seconds for suspension and 150 seconds for resumption.

In the simulator, we describe the workload as the number of operations that can be done in 168 hours, which could be calculated by (1) as: $168 * 60 * 60 * 0.2415 * x^{1.0958}$ where x = 100. To cover many pricing patterns we chose different starting times from us-east-1.linux.c1.medium spot instance's price history: 2010-01-02, 2010-09-09, and 2011-05-01. These days are selected to span different variations of *inter-price* Bands. However, we verified that running the job on other days, within the same epochs, behaves the same with a slight difference in the price and total run time. The bids range from 0.057 to 0.063 while probability density function, of the history prices of us-east-1.linux.c1.medium, shows that 99.64% of the prices fall within this range.

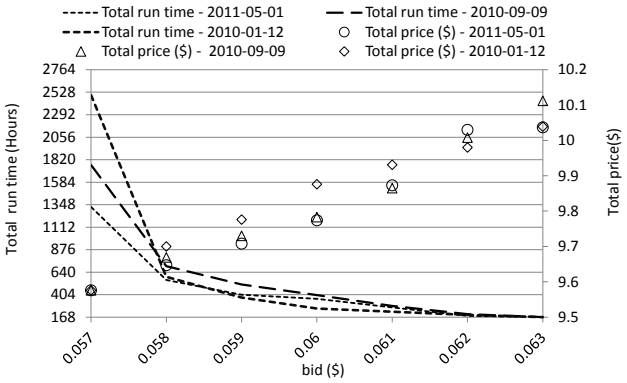

Fig. 3. Spot Instance running 168 hours job

Fig. 3 shows the following concepts: First, low bids lead to lower price but longer run time. Second, high bids lead to higher price but shorter run time. Moreover, the simulation of SI at 2010-09-09 shows a longer run time for most bid values compared with the other simulation dates. This is because of the short *inter-price* at epoch 2 (i.e., Band 3 is 1 to 2 hours). To study the influence of the *inter-price* time on the provider profit, we sum up the partial hour for each bid value. The result value is divided by the total run time to get the percentage of *Unpaid running time*. Results are illustrated in Fig. 4.

In Fig. 4, for the simulation date 2010-01-12, the provider's

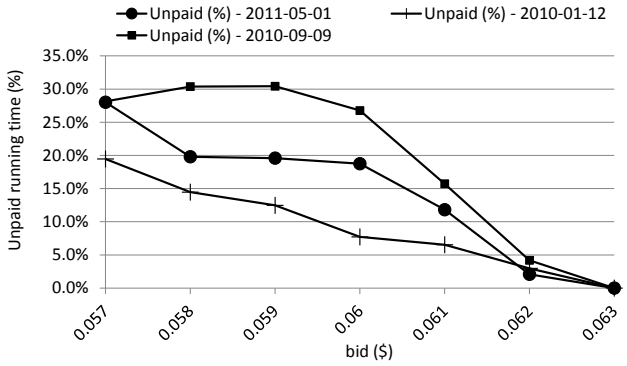

Fig. 4. Unpaid running time(%) - A Spot Instance running 168 hours job

loss (i.e., *Unpaid running time*) at low bids could be 20% of the total run time. However, the loss decreases with the higher bids values. The simulation date 2010-09-09 shows the highest loss for the provider (i.e., from 25% to 30% at low to medium bids) according to the short *inter-price* time at this epoch. Simulation at 2011-05-01 shows a reduction in provider loss compared with that done at 2010-09-09. However, it is higher than that at 2010-01-12. By this observation, we could explain the goal behind the appearance of the Bands 1 to 3 again starting from 2011-02-09. However, we cannot find any reasonable explanation behind the appearance of the Band 4 again in us-east region.

C. ESI simulation

In this section we simulate running the same described job but on ESI. The main goals of this experiment are the following: first, to observe the proposed approach consistency with the bidding concepts: 1-Low bids lead to lower price but longer run time. 2-High bids lead to higher price but shorter run time. Second, to study the proposed approach impact on the bidding strategies, and to suggest bidding strategies that boost the checkpointing technique to a level close to the optimum.

As described in Section III, when the market prices surpass the user bid, the provider reduces the VM capacity with a value proportional to the difference between user bid and the current price. However, a long run at higher price is inefficient, while it implies purchasing lower capacity with a higher price. It is user responsibility to find the best time to take a snapshot and turn off the running instance. To do that, in addition to the bid value, which is passed to the provider, the client should keep in mind another limit price value. Whenever the market price exceeds this limit, the user will take a checkpoint then terminate the instance.

We chose the same days of "US-East, Linux, c1.medium" instance price history to cover different pricing patterns as in the last experiment. The x-axis in Fig. 5(a) to Fig. 5(f) is the limit of the instance price. It is determined by the user to avoid long running of the instance at a high price. We assume that the user will start checkpointing process once the spot market price exceeded this limit value. In our simulator, it is implemented as 2 minutes delay for having a checkpoint then turning off the VM. User will be charged for this running time. Moreover, we consider that the job execution is paused during checkpointing time.

As well as the bid value, the limit value significantly affect the total price and total run time, as seen in Fig. 5. For example, in Fig. 5(a) and Fig. 5(d), if the user bid is 0.057 and the limit value is 0.058, then the total cost for running the job is 9.726$, while the total run time is 1988 hours and 12 minutes. For the same bid, if the limit value is raised to 0.059, the total cost will increase to 9.821$, while the total run time decreases to 1881 hours and 30 minutes. On the other hand, the same total price could be achieved by the bid value 0.058 and limit value 0.059 with a significant reduction in the total run time (i.e., time reduced from 1881 hour and 30 minutes to 528 hours and 18 minutes). This leads us to conclude that the optimal bid and limit values are those which satisfy the following relation: $Limit = Bid + 0.001$. It is consistent with the bidding concepts shown at the beginning of this section. Moreover, Fig. 5 shows that biding according to the relation $Limit = Bid + 2 * 0.001$ could be a good strategy, especially for average values (e.g., bid: 0.059 and limit: 0.061). However, this behavior is not consistent with all bid values because it depends on the prices distribution.

To compare the performance of the ESI with the SI, we consider Optimal checkpointing strategy as a reference. For the three simulation dates, we select the lowest bid (i.e., 0.057), the mean bid value (i.e., 0.060), and the highest bid value (i.e., 0.063). In our comparison, we consider two metrics, the total price and the total run time, where the lower normalized $Price \times Time$ is the better.

TABLE I
NORMALIZED $Price \times Time$ FOR EXECUTION ON MIN-BID, MEAN-BID, AND MAX-BID. THE REFERENCE IS A SI WITH OPTIMAL CHECKPOINTING STRATEGY SHOWN IN FIG. 3

	low-bid (0.057)	mean-bid (0.060)	max-bid (0.062)
2010-01-12	1.016 × 0.796	1.026 × 0.944	1.054 × 1.000
2010-09-09	1.026 × 0.812	1.050 × 0.797	1.047 × 1.000
2011-05-01	1.016 × 0.737	1.042 × 0.817	1.054 × 1.000

If we compare the results in Table I with what obtained by [7], the lowest normalized $Price \times Time$ of the instance us-east.c1.medium with low bid (i.e., 0.058) was 1.266. However, with our approach, even for a lower bid value (i.e., 0.057), the normalized $Price \times Time$ values were 0.809, 0.833, and 0.749 for the simulation dates in consequence, which means 34% to 40% reduction in normalized $Price \times Time$. Moreover, for the mean bid value (i.e., 0.060), the lowest normalized $Price \times Time$ of the same instance was 1.332. However, with our approach it is reduced to 0.969, 0.837, and 0.851 for the three simulation dates in consequence, which means 27% to 37% reduction in normalized $Price \times Time$.

Finally, we should remind that the *Unpaid running time* in case of ESIs is zero, which means that the provider will not lose any computation power.

 ISBN: 978-1-61208-216-5

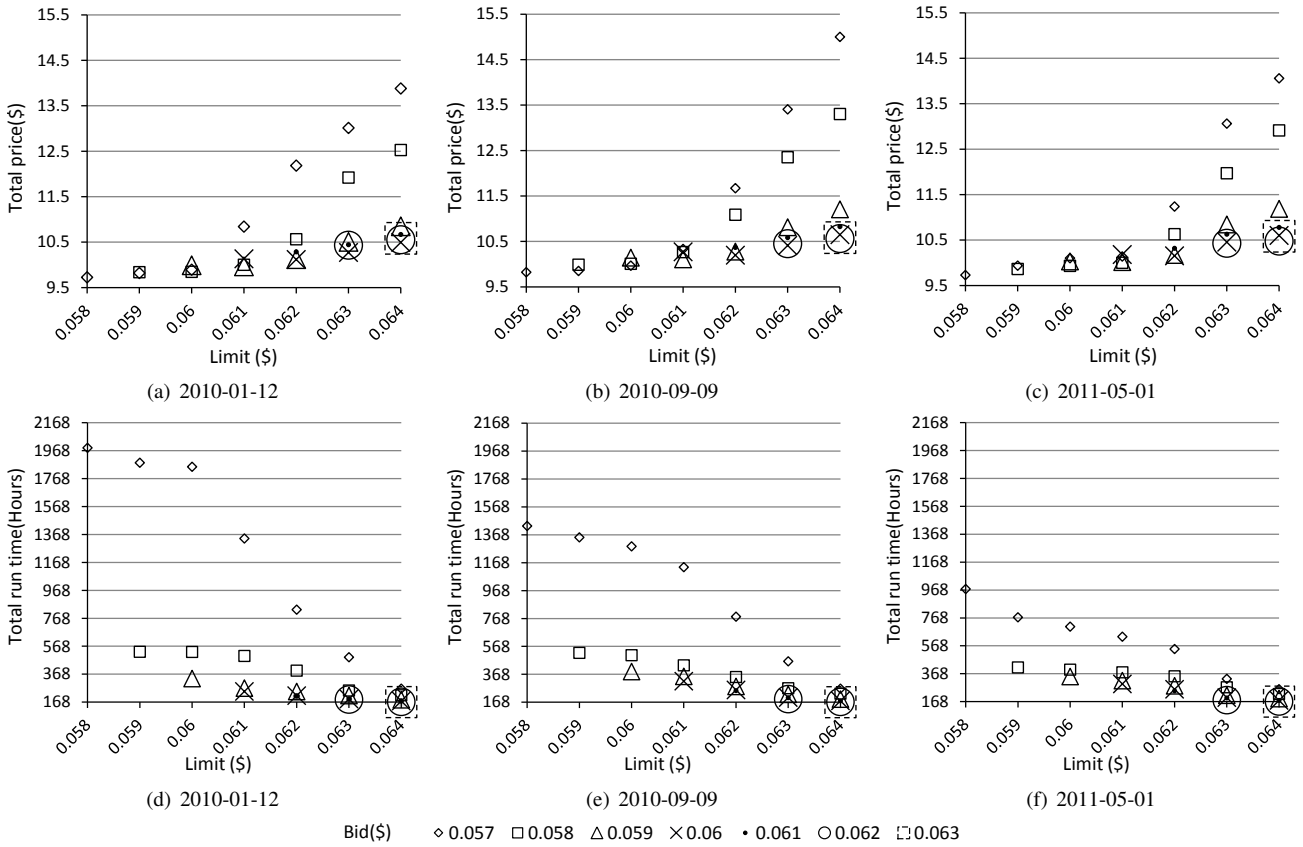

Fig. 5. Running 168 hours job on ESI with different limit prices

V. ESIs' INFLUENCE ON THE OTHER SERVICES' PERFORMANCE

In the following section, we study the influence of the ESIs on the other hosted instances (i.e., *On-Demand* and *Reserved* instances). As shown in Section IV-C, when the provider is overloaded, ESIs users may pay more money for fewer resources. It is a strong reason for ESIs users to terminate their instances, which free more resources at provider side. However, the provider should be aware of the users who choose high limit values or never attempt to terminate ESIs according to misunderstanding of ESIs concept.

In our analysis, we assume that *On-Demand* instances are hosted together with ESIs on Xen Hypervisor running *Credit Scheduler*. We started by running one *On-Demand* instance with n ESIs to understand the influence of ESIs on the performance of *On-Demand* instances. All instances are running two virtual cores. The workload is the CPU-intensive workload described in Section IV. *On-Demand* instances run with full capacity. However, ESIs' capacity is started with full capacity (i.e., 100%), then is reduced 10 percent with each step. The throughput of the *On-Demand* instance is measured with each reduction in the capacity.

As shown in Fig. 6, the *On-Demand* instance throughput is a function of both the number of ESIs and the capacity of each instance. By analyzing the curves in Fig. 6, we can notice three cases of *On-Demand* instance throughput: First, no throughput degradation. Second, throughput as a function of number of co-located ESIs only. Third, throughput as a function of number and capacity of co-located ESIs.

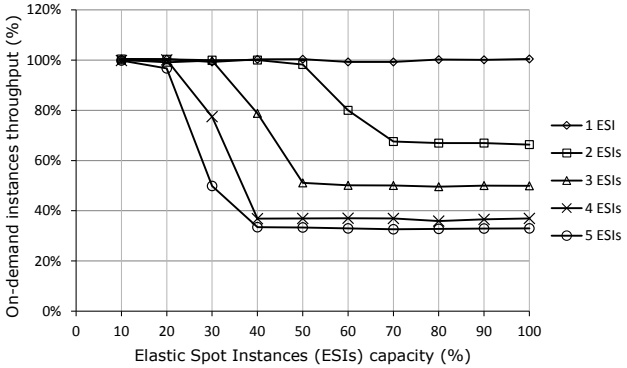

Fig. 6. *On-Demand* instances utilization with variant number of ESIs

To generalize the relation, assume that we are running n ESIs on the same host with m *On-Demand* instances. We would like to determine the capacity of the ESIs that reduce the influence of the ESI on the other hosted instances (i.e., *On-Demand* instance in our example). To formalize the models shown in Fig. 6, we define the following parameters:

- Free capacity on the host: C^h_{free}
- Number of *On-Demand* instances: n

 ISBN: 978-1-61208-216-5

- Requested capacity by *On-Demand* instance: C^d_{req}
- Assigned capacity to *On-Demand* instance: $C^d_{assigned}$
- Number of ESIs: m
- Requested capacity by ESI i: C^{si}_{req}
- Total requested capacity by ESIs: $\sum_{i=0}^{m} C^{si}_{req}$
- Assigned capacity to ESI i: $C^{si}_{assigned}$
- Total assigned capacity to ESIs: $\sum_{i=0}^{m} C^{si}_{assigned}$

In our case, we consider that the *On-Demand* instance will consume the full capacity of the CPU, so we will consider $C^d_{req} = 100$. According to our observations, we consider that a provider hosts VMs instances of the same size on one physical host. Therefore, the number of virtual cores is the same for both *On-Demand* and ESIs. In the following analysis, to isolate the other *On-Demand* instances influence, we consider hosting only one *On-Demand* instance on the physical host. However, the ESIs' impact will be the same for each *On-Demand* instance hosted on the same physical server. To calculate the allocated capacity for the *On-Demand* instance, we consider the following cases:

Case 1: The host is able to fulfill *On-Demand* instance required capacity while

$$(C^h_{free} - C^d_{req}) >= \sum_{i=0}^{m} C^{si}_{req} \quad (2)$$

In this case, the ESIs do not influence the *On-Demand* instance performance, and the *On-Demand* instance throughput is very close to 100%.

Case 2: The ESIs requested capacity is high to a level that influences the *On-Demand* instance's assigned capacity. However, the average requested capacity by an ESI is still lower than the requested capacity by *On-Demand* instance; this could be formulated as the following:

$$((C^h_{free} - \sum_{i=0}^{m} C^{si}_{req}) > C^d_{req}) \&\& ((\sum_{i=0}^{m} C^{si}_{req})/m < C^d_{req}) \quad (3)$$

In this case the *On-Demand* instance throughput is calculated by (1). Where $x = C^d_{assigned} = (C^h_{free} - \sum_{i=0}^{m} C^{si}_{req})$

Case 3: The ESIs requested capacity is very high. Moreover, the average requested capacity by an ESI is higher than or equal to the requested capacity by an *On-Demand* instance. In this case, the *Credit Scheduler* will employ its fairness to give the same capacity for each running instance on the hypervisor. In this case, the *On-Demand* instance throughput is calculated by (1), where $x = C^h_{total}/(n+m)$ and $n = 1$ in our example.

A region overloading will be reflected as high increase in the SIs' price, probably double the price of an On-Demand instance. In such a case, the ESIs should be scaled down as described in Algorithm 1 to satisfy the condition at (2). For example, if one *On-Demand* instance with two virtual cores running on the same host with 5 ESIs each with 2 cores, limiting the ESI's capacity to 20% will isolate any influence of the SIs. However, to prevent negative values of capacities in case of very high increase in the prices, we determine a static limit that cannot be exceeded. In our experiment, it was 10% of the CPU capacity. This value implies that ESIs will not influence the other hosted instances until the number of ESIs exceeds 20 instances on the same physical host.

Finally, we should remind that a very high price and low allocated capacity at overloaded time is a good reason for the customers of ESIs to have a checkpoint or/and turn off instances safely, which reduces ESIs's number and consequently reduces their influence.

VI. Related work

In this section, we classify the research towards improving the trades off between the total price, the reliability, and the total run time of SIs into two categories: first, work directed to find the best bid prices by analyzing and modeling prices history statistically (i.e., modeling price history). Second, work directed to manage SIs interruption by using fault tolerant architectures (i.e., managing the interruption). Moreover, some researches from the second category integrate history analysis techniques with fault tolerant architectures for a better performance.

A. Modeling price history

Javadi et al. [1] analyzed SIs history in terms of Spot price, *inter-price* time, but not the user bid. Andrzejak et al. [6] have proposed probabilistic decision model that considers user bid, budget, and the job deadline. The proposed model can suggest a bid value that meets a given budget or a deadline considering a specified level of confidence. Zhang et al. [9] adopted an auto-regressive model (AR) that depends on the historical values of demand to predict the next price of an instance type.

However, actual demand can neither be disclosed by the provider nor be inferred from the current price. It has been shown by Mazzucco et al. [22] that there is no correlation between SI prices and the time. Moreover, as shown in Fig. 1, the artifact changes in the SIs price make it difficult to build consistent models that describe the SIs' market behavior for the long run.

B. Managing interruption

Although MapReduce is designed as a fault tolerant architecture, it cannot tolerate a potential massive failure of instances in the SI's market. Therefore, Liu [23] extended the Cloud MapReduce (CMR) [24] implementation of map phase to stream intermediate results to a Cloud storage (i.e., SimpleDB). Their MapReduce implementation supports partial commit to keep track of the map process. In case of failure, system is able to determine the location at which the next map task should resume processing. Mattess et al. [25] examined many polices to run a Grid workload from DAS-2 [26]. Their local cluster is integrated with SI's market to cope with workload spikes, which results in reduction in the prices without degrading the performance.

Taifi et al. [27] studied running high performance computing (HPC) applications on SIs environment. To this end, they proposed SpotMPI architecture. One of SpotMPI architecture component is checkpoint-restart (CPR) calculator. Depending

 ISBN: 978-1-61208-216-5

on price history and the estimated total processing times, the CPR component determines the best checkpointing intervals. Jain et al. [28] developed an algorithm that dynamically adapts the resource allocation policy, which decides between *On-Demand* or SIs allocation. The allocation policy is adapted by learning from system performance on prior job execution while incorporating history of Spot prices and workload characteristics. However, the uncertainty of job time estimation is one of the problems that could approach [27] and [28] algorithms.

Yi et al. [7] employed checkpointing and migration as fault tolerance techniques. They examined many checkpointing strategies on the light of normalized $Price \times Time$ for different bid values and different types of instances. Moreover, after each instance's interruption, their approach decides the new SI's type, location, and price that reduces the total running time. However, as seen in Section IV-C, our approach showed outperforming results compared with their approach.

In addition to checkpointing and migration techniques, Voorsluys et al. [11] integrated job duplication technique. This integration increases the probability that jobs finish within their deadlines. However, as concluded by the authors, job duplication yields much higher costs.

VII. Conclusion and Future work

The proposed ESIs architecture does not require many modifications to the current Cloud Computing Infrastructure. However, it has benefits for both of the provider and the customer. On the provider side, our approach increases the provider's revenue where it eliminates the concept of the partial hours. For the customer, the proposed approach boosts the checkpointing strategy to the optimal level. However, clients' applications should be aware of our proposed bidding strategies.

We evaluated our approach against CPU-intensive applications where the CPU is the real player in power consumption. However, in the future, we will consider other resources and different combinations of the real workload. We will implement the techniques that work on I/O isolation like SEDF-DC. Furthermore, our extended work will include evaluating ESIs approach with the other SIs types.

References

[1] B. Javadi and R. Buyya, "Comprehensive Statistical Analysis and Modeling of Spot Instances in Public Cloud Environments," The University of Melbourne, Melbourne, Tech. Rep., 2011.

[2] R. Lämmel, "Google's MapReduce programming model; Revisited," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 208–237, Oct. 2007.

[3] Amazon, "Amazon EC2 Spot Instances." [Online]. Available: http://aws.amazon.com/ec2/spot-instances/, Retrieved: May, 2012

[4] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 75–86.

[5] S. Yi, D. Kondo, and A. Andrzejak, "Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud," in *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, Jul. 2010, pp. 236–243.

[6] A. Andrzejak, D. Kondo, and S. Yi, "Decision Model for Cloud Computing under SLA Constraints," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, Aug. 2010, pp. 257–266.

[7] S. Yi, A. Andrzejak, and D. Kondo, "Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances," *IEEE Transactions on Services Computing*, Jul. 2011.

[8] Amazon, "Amazon EC2." [Online]. Available: http://aws.amazon.com/ec2/, Retrieved: May, 2012

[9] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao, "Dynamic resource allocation for spot markets in clouds," p. 1, Mar. 2011.

[10] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and T. Dan, "Deconstructing Amazon EC2 Spot Instance Pricing," Technion Israel Institute of Technology, Haifa, Tech. Rep., 2011.

[11] W. Voorsluys and R. Buyya, "Reliable Provisioning of Spot Instances for Compute-intensive Applications," *Computing Research Repository*, vol. abs/1110.5, 2011.

[12] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance." in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 101–110.

[13] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 workshop on Experimental computer science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007.

[14] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance VMM-bypass I/O in virtual machines," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006, p. 3.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, *Xen and the art of virtualization*. New York, New York, USA: ACM Press, Oct. 2003, vol. 37, no. 5.

[16] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, Sep. 2007.

[17] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in Xen," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, ser. Middleware '06. New York, NY, USA: Springer-Verlag New York, Inc., 2006, pp. 342–362.

[18] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[19] T. Lossen, "Cloud exchange." [Online]. Available: http://cloudexchange.org/, Retrieved: May, 2012

[20] NASA, "NAS Parallel Benchmarks (NPB)." [Online]. Available: http://www.nas.nasa.gov/Resources/Software/npb.html, Retrieved: May, 2012

[21] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Resource Leasing and the Art of Suspending Virtual Machines," *High Performance Computing and Communications, 10th IEEE International Conference on*, vol. 0, pp. 59–68, 2009.

[22] M. Mazzucco and M. Dumas, "Achieving Performance and Availability Guarantees with Spot Instances," in *13th International Conference on High Performance Computing and Communications (HPCC-2011)*, Banff (Canada).

[23] H. Liu, "Cutting MapReduce Cost with Spot Market," in *USENIX HotCloud'11*, Portland, USA, 2011, p. 5.

[24] H. Liu and D. Orban, "Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 464–474, 2011.

[25] M. Mattess, C. Vecchiola, and R. Buyya, *Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market*. IEEE, Sep. 2010.

[26] Das-2, "The Distributed ASCI Supercomputer 2 (DAS-2)." [Online]. Available: http://www.cs.vu.nl/das2/, Retrieved: May, 2012

[27] M. Taifi, J. Shi, and A. Khreishah, "SpotMPI: A Framework for Auction-Based HPC Computing Using Amazon Spot Instances," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds. Springer Berlin / Heidelberg, 2011, vol. 7017, pp. 109–120.

[28] N. Jain, I. Menache, and O. Shamir, "On-demand or Spot? Learning-Based Resource Allocation for Delay-Tolerant Batch Computing," in *Microsoft Research*, 2011, p. 10.

 ISBN: 978-1-61208-216-5