Deconstructing Amazon EC2 Spot Instance Pricing

Orna Agmon Ben-Yehuda Muli Ben-Yehuda Assaf Schuster Dan Tsafrir

Technion – Israel Institute of Technology
{ladypine, muli, assaf, dan}@cs.technion.ac.il

***Abstract*—Cloud providers possessing large quantities of spare capacity must either incentivize clients to purchase it or suffer losses. Amazon is the first cloud provider to address this challenge, by allowing clients to bid on spare capacity and by granting resources to bidders while their bids exceed a periodically changing spot price. Amazon publicizes the spot price but does not disclose how it is determined.**

By analyzing the spot price histories of Amazon's EC2 cloud, we reverse engineer how prices are set and construct a model that generates prices consistent with existing price traces. We find that prices are usually not market-driven as sometimes previously assumed. Rather, they are typically generated at random from within a tight price interval via a dynamic hidden reserve price. Our model could help clients make informed bids, cloud providers design profitable systems, and researchers design pricing algorithms.

***Keywords*-cloud; spot price; spot instance; Amazon EC2**

I. Introduction

Unsold cloud capacity is wasted capacity, so cloud providers would naturally like to sell it. Clients might be enticed to purchase it if they are provided with enough incentive, notably, a cheaper price. In late 2009, Amazon was the first cloud provider to attempt to provide such an incentive by announcing its *spot instances* pricing system. "Spot Instances [...] allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current Spot Price. The Spot Price changes periodically based on supply and demand, and customers whose bids exceeds it gain access to the available Spot Instances" [1]. With this system, Amazon motivates purchasing cheaper capacity while ensuring it can continuously act in its best interest by maintaining control over the spot price. **Section II** summarizes the publicly available information regarding Amazon's pricing system.

Amazon does not disclose its underlying pricing policies. Despite much interest [2]–[4], Amazon's spot pricing scheme has, until now, remained largely unknown. The only information that Amazon does reveal is its temporal spot prices, which must be publicized to make the pricing system work. While Amazon provides only its most recent price history, interested parties record and accumulate all the data ever published by Amazon, making it available on the Web [5], [6]. We leverage the resulting trace files for this study. The trace files, along with the methodology we employ to use them, are described in **Section III**.

Knowing how a leading cloud provider like Amazon prices its unused capacity is of potential interest to both cloud providers and cloud clients. Understanding the considerations, policies, and mechanisms involved may allow other providers to better compete and to utilize their own unused capacity more effectively. Clients can likewise exploit this knowledge to optimize their bids, to predict how long their spot instances would be able to run, and to reason about when to purchase cheaper or costlier capacity.

Motivated by these benefits, we attempt in **Sections IV–V** to uncover how Amazon prices its unused EC2 capacity. We construct a spare capacity pricing model and present evidence suggesting that prices are typically *not* determined according to Amazon's public definition of the spot pricing system as quoted above. Rather, spot prices are usually drawn from a tight, fixed price interval, reflecting a random reserve price that is not driven by supply and demand (A *reserve price* is a hidden price below which bids are ignored). Consequently, published spot prices reveal little information about actual, real-life client bids; studies that assume otherwise (e.g., [7], [8]) are misguided and their results questionable. We speculate that Amazon utilizes such a price interval because its spare capacity usually exceeds the demand.

In **Section VI** we put the model that we have developed to the test by conducting pricing simulations and by showing their results are consistent with EC2 price traces. We then discuss the possible benefits of using reserve price systems (such as the one apparently used by Amazon) in **Section VII**. Finally, we survey the related work in **Section VIII** and offer some concluding remarks in **Section IX**.

II. Pricing Cloud Instances

Amazon's EC2 clients rent virtual machines called *instances*, such that each instance has a *type* describing its computational resources as follows: m1.small, m1.large and m1.xlarge, respectively denote small, large, and extra large "standard" instances; m2.xlarge, m2.2xlarge, and m2.4xlarge respectively denote extra large, double extra large, and quadruple extra large "high memory" instances; and c1.medium and c1.xlarge respectively denote medium and extra-large "high CPU" instances.

An instance is rented within a geographical *region*. We use data from four EC2 regions: us-east, us-west, eu-west and

ap-southeast, which correspond to Amazon's data centers in Virginia, California, Ireland, and Singapore, respectively.

Amazon offers three purchasing models, all requiring a fee from a few cents to a few dollars, per hour, per running instance. The models provide different assurances regarding when instances can be launched and terminated. Specifically, paying a yearly fee (of hundreds to thousands of dollars) buys clients the ability to launch one *reserved instance* whenever they wish. Clients may instead choose to forgo the yearly fee and attempt to purchase an *on-demand instance* when they need it, but then the hourly fee is a bit higher and there is no guarantee that launching will be possible at any given time. Both reserved and on-demand instances remain active until they are terminated by the client.

The third, cheapest purchasing model provides no guarantee regarding both launch- and termination-time. When placing a request for a *spot instance*, clients bid the maximum hourly price they are willing to pay for running it (called *declared price* or *bid*). The request is granted if the bid is higher than the spot price, otherwise it waits. Periodically, Amazon publishes a new *spot price* and launches all waiting instance requests with a maximum price exceeding this value; the instances will run until clients terminate them or the spot price increases above their maximum price. All running spot instances incur a uniform hourly charge, which is the current spot price. The charge is in full hours, unless the instance was terminated due to a spot price change, in which case the last hour fraction is free of charge.

In this work, we assume that instances with bids equal to the spot price are treated similarly to instances with bids higher than the spot price.

III. Methodology

Trace Files: We analyze 64 ($= 8\times4\times2$) spot price trace files associated with the 8 aforementioned instance types, the 4 aforementioned regions, and 2 operating systems (Linux and Windows). The traces were collected by Lossen [5] and Vermeersch [6]. They start as early as November 30, 2009 (traces for region ap-southeast are only available from the end of April 2010). In this paper, unless otherwise stated, we use data accumulated until July 13, 2010,

Availability: A *persistent request* is a series of requests for an instance that is immediately re-requested every time it is terminated due to the spot price rising above its bid. Given a declared price D, we define D's *availability* to be the time fraction in which a persistently requested instance would run if D is its declared price. Formally, let F be a spot price trace file, and let T_b and T_e be the beginning and end of a time interval within F. The availability of D within F during $[T_b, T_e]$ is:

$$availability\ {}^{F}_{b\to e}(D) \;=\; \frac{T^F_{b\to e}(D)}{T_e - T_b}$$

where $T^F_{b\to e}(D)$ denotes the time between T_b and T_e during which the spot price was lower than or equal to D; namely, if $\{T_i\}_{i=0}^{N}$ denote times at which the spot price changed ($T_b = T_0$ and $T_e = T_N$) and S_i denotes the spot price during the time interval $[T_{i-1}, T_i]$ ($i = 1, 2, ..., N$), then

$$T^F_{b\to e}(D) \;=\; \Sigma_{\{i|S_i\le D\}}\left(T_i - T_{i-1}\right).$$

We analyze availability as a function of the declared price D, noting that it reflects the probability that the corresponding spot instances would be immediately launched when requested at some uniformly random time within $[T_b, T_e]$.

IV. Evidence for Artificial Pricing Intervention

A. Market-Driven Auction

Amazon's description of "How Spot Instances Work" [1] gives the impression that spot prices are set through a uniform price, sealed-bid, market-driven auction. "Uniform price" means all bidders pay the same price. "Sealed-bid" means bids are unknown to other bidders. "Market-driven" means the spot price is set according to the clients' bids. One example of such an auction is an $(N+1)^{th}$ price auction (VCG) [9]–[11] of multiple goods, with retroactive supply limitation (after clients bid), to maximize the provider's revenue. However, Amazon might be using some other mechanism that is consistent with their description.

In an $(N+1)^{th}$ price auction of multiple goods, each client bids for a single good (i.e., a spot instance). The provider chooses the top N bidders. The provider may set N up-front on the basis of available capacity. A *greedy provider* might retroactively set N after receiving the bids, to maximize revenue. In any case, N cannot exceed the total number of goods. The provider sets the uniform price to the price declared by the highest bidder who *did not* win the auction (bidder number $N+1$) and publishes it. The top N winning bidders pay the published price and their instances start running. In this case, the published price is a price bid by an actual client.

The provider may also decide to ignore bids below a hidden *reserve price* or below a publicly known *minimal price*, to prevent the goods from being sold cheaply, or to give the impression of increased demand.

We conjecture that most of the time, contrary to impressions conveyed by Amazon [1] and assumptions made by researchers [7], [8], the spot price is set according to a constantly changing reserved price, disregarding client bids. In other words, most of the time the spot price is *not* market-driven but is set by Amazon according to an undisclosed algorithm.

B. Evidence: Availability as a Function of Price

In support of this conjecture, we analyze the relationship between an instance's declared price (how much a client would be willing to pay for it) and the resulting availability between January 20th and July 13th, 2010. Figure 1 shows

the availability of different spot instance types as a function of declared price (price-availability graphs), for all examined Windows spot instance types in all regions. Results for instances running Linux (not shown) are qualitatively similar. The prices of different resources seem unrelated, except that they share the same functional shape: a sharp linear increase in availability until a "knee" (sharp change in slope) is reached. The knee is usually high, representing an availability of 0.95 or more. After the knee, the availability grows with declared price but at a slower, non-linear rate.

Figure 2 shows *normalized* price-availability graphs for Linux: prices on the horizontal axis are normalized by their instance-type's respective on-demand prices. We see that Linux types can be classified by region. Each of the two region classes has a distinct normalized price range in which the availability's dependency on the price is linear. One class contains us-east, and the other class contains other regions.

Figure 3 shows the data presented in Figure 1 as normalized price-availability graphs. As in Figure 2, different types can be classified by region: us-east or all other regions. Not as in Figure 2, different types have different normalized prices within a class, and the relative price difference between any type pair is the same in each class. The m1.small type, presented in the figures by red circles, has a particularly low knee, with an availability of 0.45. Figures 1–3 show that availability strongly depends on declared price for all regions and all instance types, and that this dependency has a typical recurring shape. The similarity between different regions can be explained if we assume that Amazon uses the same mechanism to set the price in different regions. The recurring shaped dependency could be explained in one of two ways: either Amazon's spot prices reflect real client bids and the shaped dependency occurs naturally, or the spot prices are the result of an artificial dynamic hidden reserve price algorithm, of which the shaped dependency is an artifact.

Let us first assume that the shaped dependency occurs naturally due to real client bids. The differences between absolute price ranges of the same type in different regions (Figure 1) show that different regions experience different supply and demand conditions. This means that uncoordinated client bids for different types and regions, which experience different supply and demand conditions, would have to naturally and independently create *all* of the following phenomena: (1) normalized prices turning out identical for various Linux types but different for Windows types; (2) a rigid linear connection between availability and price that turns out identical for different types and regions; (3) a singular region having a normalized price range different than all the rest (which turn out to have identical ranges); and (4) normalized prices for Windows instances which differ from one another by identical amounts in each region class, creating the same pattern for both region classes.

Our hypothesis we consider it unlikely that all four phenomena could have resulted from Amazon setting the price solely based on client bids, and therefore we lean toward the following hypothesis: that Amazon uses a dynamic algorithm to set a reserve price for the auction, ignoring client bids. Most of the time, the auction's result is identical to the reserve price. Thus most of the time the prices Amazon announces are not market-driven. Both the simulation results presented in Section VI and Occam's razor—preferring the simplest explanation—support this hypothesis.

We contend that the artificial reserve price algorithm gets as input a *floor price* and a *ceiling price* for each spot instance type, with the floor and ceiling prices expressed as fractions of the on-demand price. The floor price is the minimal price. The ceiling price is the price corresponding to the knee in the graph, or the maximal price if no knee exists. We refer to this price range, in which availability is a linear function of the price, as the pricing *band*. The algorithm then changes the reserve price using a random process, such that there is a linear relation between availability and prices in the floor–ceiling range. It guarantees that the reserve price never drops below the floor, which reflects Amazon's minimal-reserve price for spot instances, nor rises above the ceiling. Thus, many of the price changes seen in Amazon's EC2 price traces are not evidence of demand or supply changes, but rather evidence of an artificial decision to change the reserve price. Amazon sets different floor and ceiling prices for the us-east region, for different sets of types, and for different operating systems. The random process's role is probably to generate a false sense of dynamic market conditions, as discussed in Section VII.

C. Random Reserve Price

We deconstruct the reserve price algorithm using traces from April–July, 2010, when the spot price in eight ap-southeast.windows instance types almost always stayed within the artificial band. We matched the price changes in those traces (denoted by Δ) with an $AR(1)$ (auto-regressive) process . We found a good match to the following process:

$$\Delta_i = -a_1 \Delta_{i-1} + \epsilon(\sigma), \tag{1}$$

where $a_1 = 0.7$ and $\epsilon(\sigma)$ is white noise with a standard deviation σ. Let F, C denote the floor and ceiling of the artificial band, respectively. We matched σ with a value of $0.39(F - C)$. These parameters fit all the analyzed types except m1.small, which matched different values ($a_1 = 0.5, \sigma = 0.5(F - C)$). The standard deviations are given in Figure 4. This close fit—the same parameters characterizing the randomness of several different traces (except m1.small)—is consistent with our hypothesis that the prices are usually set by an artificial algorithm.

Prices within the band might also result from clients bidding within the band (although others have already noted that such bids are not cost-effective [2], [4]), but mean price analysis indicates otherwise. Since an $AR(1)$ process

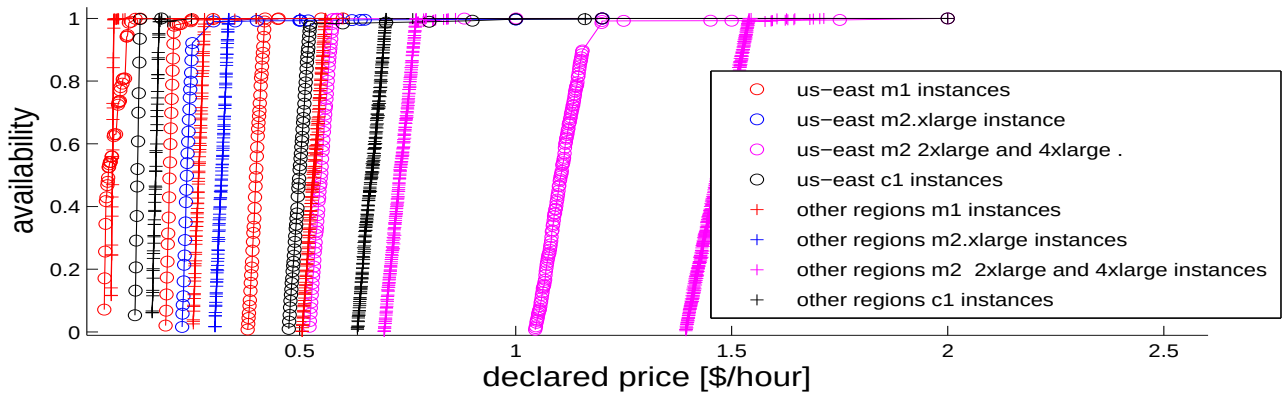

Figure 1: Availability of Windows-running spot instance types as a function of their declared price. The legend is multiplexed: us-west, eu-west, ap-southeast all appear in the legend as "other regions". m1.small, m1.large and m1.xlarge all appear as m1. c1.medium and c1.xlarge appear as c1.

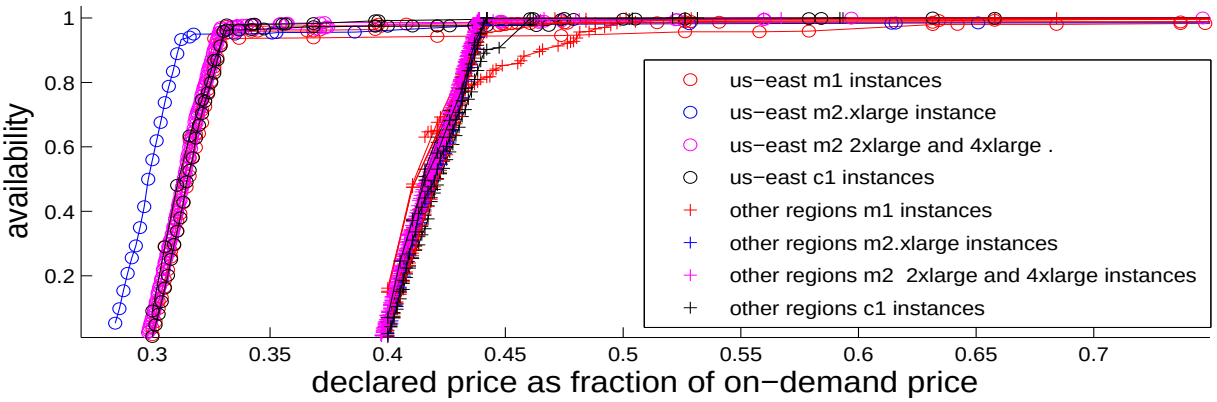

Figure 2: Availability of Linux-running spot instance types as a function of their normalized declared price. The declared price is divided by the price of a similar on-demand instance. The legend is multiplexed as in Figure 1. Most of the curves overlap.

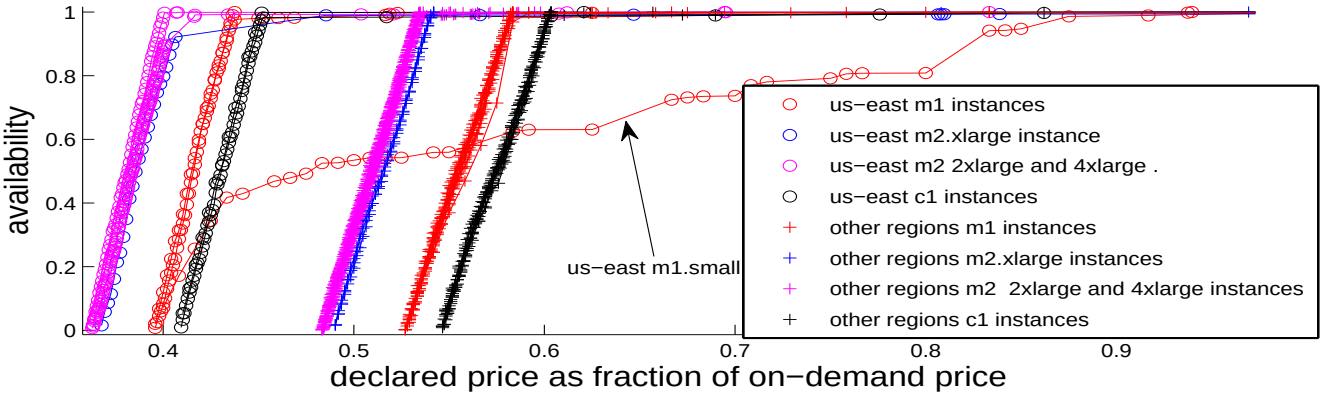

Figure 3: Availability of Windows-running spot instance types as a function of their normalized declared price. The declared price is divided by the price of a similar on-demand instance. The legend is multiplexed as in Figure 1. Many of the curves overlap. us-east.windows.m1.small is indicated by an arrow.

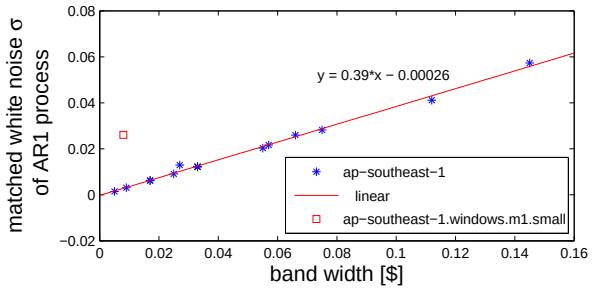

Figure 4: Standard deviation of matched $AR(1)$ process as a function of artificial price band width.

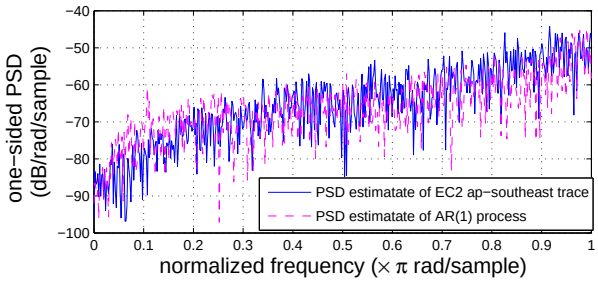

Figure 5: Power Spectral Density (Periodogram) estimate of an EC2 price trace, and our derived $AR(1)$ price trace.

is symmetric, its theoretic average is the middle of the band. For the 8 traces we used here, the average price was 98%-100% of the middle of the band. This, too, supports our hypothesis that the spot price within the band is almost always determined solely by the $AR(1)$ process, i.e., is equal to the reserve price. In addition, we find that on average over all the 64 traces we analyzed, prices are within the band 98% of the time. We conclude that prices are determined artificially by an $AR(1)$ reserve price algorithm and do not represent real client bids around 98% of the time.

On the basis of this analysis, we construct the $AR(1)$ *reserve price algorithm*: The first two reserve prices are defined as $P_{-1} = F + 0.1(C - F)$, $P_0 = F$. The following prices are defined as $P_i = P_{i-1} + \Delta_i$, where $\Delta_i = -0.7 * \Delta_{i-1} + \epsilon(0.39 * (C - F))$. The process is truncated to the $[F, C]$ range by regenerating the white noise component while P_i is outside the $[F, C]$ range or identical to P_{i-1}. All prices are rounded to one-tenth of a cent, as done by Amazon during 2010.

Figure 5 provides a spectral analysis of one of the Amazon traces and of prices produced by our $AR(1)$ algorithm. The match of frequencies shows that our reverse-engineered reserve price algorithm is consistent with Amazon's algorithm.

V. Pricing Epochs

To statistically analyze spot price histories, it would be erroneous to assume that the same pricing model applies to all the data in the history trace. Rather, each trace is divided to contiguous epochs associated with different pricing policies. We show here that our main traces are divided into three parts as depicted in Figure 6. Since the pricing mechanism changes significantly between epochs, data regarding these epochs should be separated if an associated statistical analysis is to be sound. Accordingly, for the purpose of evaluating the effectiveness of client algorithms, strategies, and predictions, the data from a (single) epoch of interest should be used.

The *first epoch* starts, according to our analysis, as early as November 30th, 2009 and ends on December 14th, 2009, the date on which Amazon announced the availability of spot instances. During this time instances were unknown to the general public. Hence, the population which undertook any bidding during the first epoch was smaller than the general public, of nearly constant size, and possibly had additional information regarding the internals of the pricing mechanism at that time.

The *second epoch* begins with the public announcement on December 14th, 2009. It ends with a pricing mechanism change around January 8th, 2010, when minimal spot prices suddenly change (usually decrease, but in Figure 6 we see an increase). It is characterized by long intervals of constant low prices.

The *third epoch* begins on January 20th, 2010. Instance types and regions began to change minimal price around January 8th, but we define the beginning of the epoch as the date in which the last one (eu-west.linux.m2.2xlarge) reached a new minimal price. Due to (1) the gradual move to the new minimal values and to (2) a bug in the pricing mechanism that was fixed in mid-January 2010 [12], we choose to disregard data from the transition period between the second and third epochs.

Additional epoch-defining dates are July 25th, 2010 and February 9th, 2011, in which the algorithm for timing price changes changed (see Section VI). On January 24th, 2011, in addition to the original single price band identified in Section IV-B, Amazon introduced a second distinct band in several instance types, as seen in Figure 7.

These time-coordinated abrupt changes in many regions and instance types further support our hypothesis, since prices are likely to undergo coordinated and abrupt changes precisely where they are artificial. Another interesting temporal pattern can be found in policy changes starting new epochs on a regular basis, approximately every 6 months.

VI. Spot Price Simulation

To get a better feel for the validity of our hypothesis, we simulated the prices and availability resulting from setting the price via a sealed-bid $(N + 1)^{th}$ price auction with a reserve price, where the number of supplied instances is retroactively limited to maximize provider revenue. The on-demand price was defined as 1. The reserve price was either constant (0.4) or the $AR(1)$ algorithm defined in Section IV-C, with a band of $[0.4, 0.45]$.

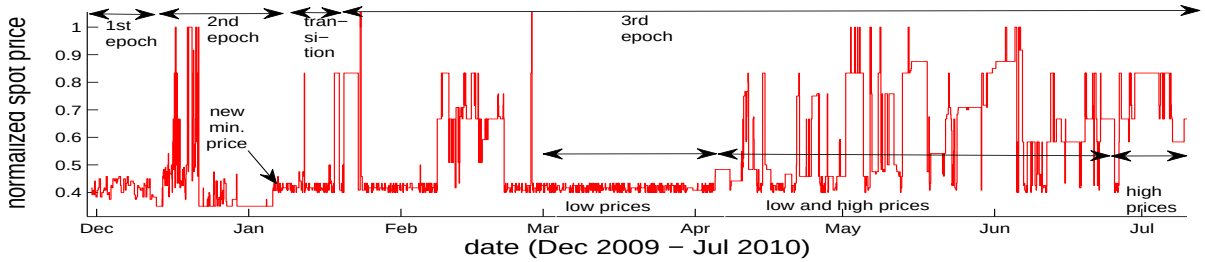

Figure 6: Price history for us-east.windows.m1.small. Three time epochs are shown, with a transition period between the second and third epochs. Spot price is presented as a fraction of the on-demand price for the same instance.

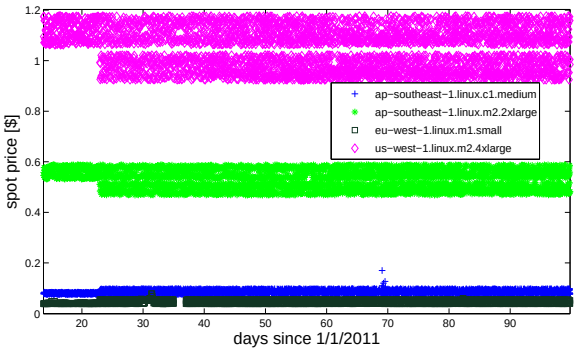

Figure 7: Double band introduction on January 24th, 2011.

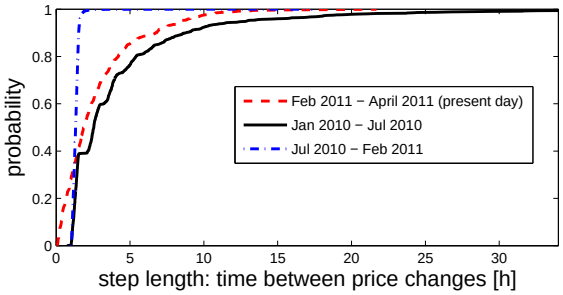

Figure 8: CDF of time interval between price changes for different versions of the price change scheduling algorithm. Input: us-east.linux.m1.small

Workload Modeling: In the absence of cloud workload traces, we fed the simulation with 20K tasks, with run-times in the range of 10 minutes to 24 hours, out of the LPC-EGEE cluster workload from 2004, kindly provided by Medernach [13] via the Parallel Workloads Archive [14]. LPC-EGEE is characterized by tasks which are small in comparison to the capacity of the cluster, allowing for elasticity. In the simulation, each task was interpreted as a single instance, submitted at the same time and requiring the same amount of run-time as in the original trace to complete.

Customer Bid Modeling: Due to the lack of information on the distribution of real client bids (since we argue that most prices do not supply such information), we compare several bidding models, and verify that the qualitative results are insensitive to the bid modeling. All the distributions were adjusted to uniform minimal and on-demand prices.

The first model is a Pareto distribution (a widely applicable economic distribution [15], [16]) with a minimal value of 0.4, and a Pareto index of 2, a reasonable value for income distribution [15]. The second model is $\mathcal{N}(0.7, 0.3^2)$, truncated at 0.4. The third is a linear mapping from runtimes to $(0.4, 1]$, which reflects client aversion to having long-running instances terminated.

Price Change Timing: Price changes in the simulation are triggered according to the CDF of intervals between price changes, collected during January–July 2010, and given in Figure 8 (solid black line). This period was characterized by quiet times—prices never changed before 60 minutes or between 90 and 120 minutes since the previous price change.

It is interesting to note that such quiet time can be monetized by clients to gain free computation power with a probability of about 25%, by submitting an instance with a bid of the current spot price 31 minutes after a price change. The instance would then have a 50% possibility of undergoing another price change within 30-60 minutes. If that change is a price increase, the instance would be terminated, and the client would gain, on average, 45 minutes of free computation. Clients do not exploit this loophole in our simulation.

Figure 8 also presents the evolution of the timing of price changes. The next algorithm (in place from July, 2010 until Feb 8th, 2011) allowed for a quiet hour after a price change. The following one (starting Feb 9th, 2011) matches an exponential distribution with a 1.5 hour rate parameter, with five quiet minutes, which is almost memory-less, and prevents abuse of the timing algorithm.

Simulator Event-Driven Loop: We created a trace-based event-driven simulator, where events are: (1) instance submission and termination and (2) price changes (due to a scheduled change or to a waiting instance with a bid higher than the spot price). We ran the simulation on 70 CPUs, according to number in the LPC-EGEE trace. We ended the simulation when the last input trace job had been submitted.

Simulation Results: Simulation results in terms of price-availability graphs are presented in Figure 9, for different bid models and price setting mechanisms. The functions of simulations with the $AR(1)$ reserve price feature a linear

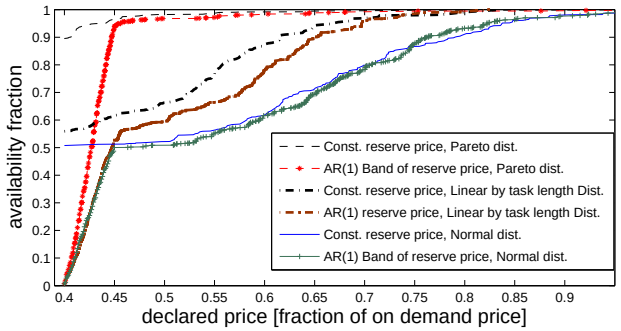

Figure 9: Simulation results for various bidding models, with constant and $AR(1)$ reserve price.

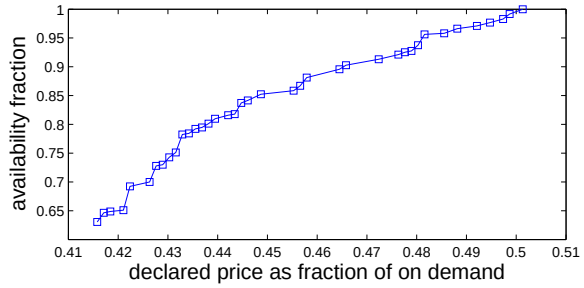

Figure 10: Availability as a function of the declared price during the second epoch for us-west.linux.m1.xlarge.

segment and a knee in high availability, as do the availability functions of EC2 during the third epoch, which are shown in Figures 1, 2, and 3. The constant reserve price functions do not exhibit this behavior. Rather, they are jittery, like the high price regime of the us-east.windows.m1.small graph in Figure 3, and the second epoch graph in Figure 10.

Furthermore, the availability of the reserve price in the constant reserve price simulations is high (0.5, 0.56, 0.9), as it is in the second epoch (0.63 in Figure 10). In contrast, the availability of the minimal price in the $AR(1)$ reserve price simulations and in the third epoch tends to zero as the number of discrete prices within the band grows.

We consider these simulation results another indication that most prices in the EC2 traces during the third epoch are set using an artificial, non-market-driven algorithm, in particular an AR(1) reserve price. The simulation results also suggest that Amazon set prices via a market-driven auction with a constant reserve price during the second epoch (December, 2009 until January, 2010), and that prices above the band are market-driven.

VII. Dynamic Reserve Price Benefits

The dynamic $AR(1)$ reserve price mechanism is a uniform price mechanism, but it is not market-driven. It has several long-term, wide-range, benefits that may justify its use. Like a constant minimal or reserve price, it guarantees that on-demand instances are not completely cannibalized by spot instances. Yet it also allows the provider to sell instances on machines which would otherwise run idle, to provide elasticity for the fixed price instances. Spot instances can be quickly evacuated, and still reduce the costs associated with idle servers, with no real harm to the main offering of on-demand instances.

Katkar and Reiley [17] found that for low-priced eBay sales of up to $20, (hidden) reserve prices deter good clients and yield lower revenues than minimal (published) prices. Nevertheless, providers can use a random algorithm to set a hidden reserve price, with no obligation to inform clients. A random reserve price is better than a constant minimal price, because it maintains an impression of constant change, thus preventing clients from becoming complacent. It forces them to either bid higher than the band or tolerate sudden unavailability. It also serves to occasionally clear queues of low bids within the band, which is not the case for a constant reserve price that is equal to the ceiling price.

A random reserve price might also serve other goals. If the public is unaware of its use, the random reserve price can mask times of low demand and of price inactivity, by creating an impression of false activity (demand and supply changes), thus possibly driving up spot prices or the provider's stock. A large enough band covering the spectrum of probable prices could also mask high demand and low supply, and thus help to maintain the illusion of an infinitely elastic cloud. However, if the artificial band is relatively small, as in the case of Amazon EC2 spot prices, the provider's use of an $AR(1)$ process for setting the price within the band is a strong indication of low demand.

VIII. Related Work

Optimizing Client Goals Using Spot Price Traces: Andrzejak, Kondo and Yi used spot price histories to advise the client how to minimize monetary costs while meeting an SLA [18], and to schedule checkpoints [19] and migrations [20]. The first two works used data from the transition period between the second and third epoch, and focused on eu-west, which suffered most from the transition. The last interchangeably used data from before and after the change in the price change algorithm on July 25, 2010.

Mattess, Vecchiola and Buyya [4] examined client strategies for using spot instances to manage peak loads.They used an EC2 spot instance trace of the third epoch only, attributing the different trace behavior prior to January 18th to Christmas and to the recent introduction of spot instances. They identified the price band, noted that bidding just above the band is almost as good as bidding very high, and recommended bidding right under the on-demand price.

Chohan et al. [2] note the cost-effectiveness of bidding at the top of the band. They analyze price histories from the third epoch only, because of a pricing bug, which was fixed in mid-January 2010 [12]. The bug allowed a region to have a low price while in at least one availability zone in that region instances with higher prices were terminated due to congestion. The authors attributed the qualitative change of

prices between the second and third epoch to the bug fix. However, this bug fix is unlikely to have caused the more significant price changes we observe during January 2010.

Optimizing Provider Algorithms Using Spot Traces: Zhang et al. [7] assumed EC2 uses hourly price changes, contrary to our findings, and a market-driven auction. As a consequence, they assumed spot price histories reflect actual client bids. On this basis, which we found is false on average 98% of the time, they sought resource allocations which optimized the provider's revenue. Chen et al. [8] assumed EC2 price traces represent market clearing prices.

Free Spot and Futures Markets: While Amazon is currently the only provider offering "spot instances," free computing resource markets have already been analyzed [21]–[23]. Price traces of such free markets [21], [22] differ from EC2 spot price traces: they do not have a hard minimal price and are not anchored in the bottom of the price range. Rahman, Lu and Gupta [24] evaluate spot options using EC2 traces, and note that the "data does not show enough fluctuations as expected in a free market."

IX. Conclusions

Amazon EC2 spot price traces provide more information about Amazon than about its clients. We have shown that Amazon sets spot prices using a hidden random $AR(1)$ reserve price in addition to a market-driven mechanism, such that high prices are set by the market-driven mechanism, but most low prices, within a band of prices, are actually set according to an artificial dynamic reserve price.

Understanding how Amazon prices its spare capacity is useful for clients, who can decide how much to bid for instances; for providers, who can learn how to build more profitable systems; and for researchers, who can differentiate between prices set by an artificial process and prices likely to have been set by real client bids. We have shown that many price trace characteristics (e.g., minimal value, band width, change timing) are artificial, and might change every six months according to Amazon's decisions. Thus, researchers should be aware of the epochs present in their traces when using those traces to model future price behavior or to evaluate client algorithm performance. We have shown that indiscriminately using Amazon's current traces to model client behavior is unfounded on average 98% of the time.

References

[1] "Amazon EC2 spot instances," http://aws.amazon.com/ec2/spot-instances/, (Accessed Aug, 2011).

[2] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: using spot instances for mapreduce workflows," in *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[3] D. Samovskiy, "Unexpected similarities in EC2 spot price history between regions," http://tinyurl.com/somic, (Accessed Apr, 2011).

[4] M. Mattess, C. Vecchiola, and R. Buyya, "Managing peak loads by leasing cloud infrastructure services from a spot market," in *IEEE Int'l conference on High Performance Computing and Communications (HPCC)*, 2010.

[5] T. Lossen, "The cloud exchange web site," http://cloudexchange.org/, (Accessed Apr, 2011).

[6] K. Vermeersch, "The spot watch web site," http://spotwatch.eu/input/, (Accessed Apr, 2011).

[7] Q. Zhang, E. Gurses, R. Boutaba, and J. Xiao, "Dynamic resource allocation for spot markets in clouds," in *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2011.

[8] J. Chen, C. Wang, B. B. Zhou, L. Sun, Y. C. Lee, and A. Y. Zomaya, "Tradeoffs between profit and customer satisfaction for service provisioning in the cloud," in *HPDC*, 2011.

[9] W. Vickrey, "Counterspeculation, auctions, and competitive sealed tenders," *Journal of Finance*, vol. 16, no. 1, 1961.

[10] E. H. Clarke, "Multipart pricing of public goods," *Public Choice*, vol. 11, no. 1, pp. 17–33, Sep 1971.

[11] T. Groves, "Incentives in teams," *Econometrica*, vol. 41, no. 4, pp. 617–631, Jul 1973.

[12] "Spot instance termination conditions?" http://tinyurl.com/2dzp734, 2011, online AWS Developer Forums discussion. (Accessed Apr, 2011).

[13] E. Medernach, "Workload analysis of a cluster in a grid environment," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2005.

[14] D. Feitelson, "Parallel workloads archive," Website, http://www.cs.huji.ac.il/labs/parallel/workload/index.html.

[15] W. Souma, "Physics of personal income," 2002. [Online]. Available: http://arxiv.org/pdf/cond-mat/0202388

[16] M. Levy and S. Solomon, "New evidence for the power-law distribution of wealth," *Physica A*, vol. 242, pp. 90–94, 1997.

[17] R. Katkar and D. H. Reiley, "Public versus secret reserve prices in ebay auctions: Results from a pokmon field experiment," *Advances in Economic Analysis and Policy*, 2006.

[18] A. Andrzejak, D. Kondo, and S. Yi, "Decision model for cloud computing under SLA constraints," in *IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[19] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2010.

[20] S. Yi, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on Amazon cloud spot instances," *IEEE Transactions on Services Computing*, 2011.

[21] F. M. Ortuno and U. Harder, "Stochastic calculus model for the spot price of computing power," in *Annual UK Performance Engineering Workshop (UKPEW)*, 2010.

[22] K. Vanmechelen, W. Depoorter, and J. Broeckhove, "Combining futures and spot markets: A hybrid market approach to economic grid resource management," *Journal of Grid Computing*, vol. 9, pp. 81–94, 2011.

[23] J. Altmann, C. Courcoubetis, G. Stamoulis, M. Dramitinos, T. Rayna, M. Risch, and C. Bannink, "GridEcon: A market place for computing resources," in *Grid Economics and Business Models*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5206, pp. 185–196.

[24] M. R. Rahman, Y. Lu, and I. Gupta, "Risk aware resource allocation for clouds," University of Illinois at Urbana-Champaign, Tech. Rep., 2011. [Online]. Available: http://hdl.handle.net/2142/25754