A Combinatorial Auction-Based Mechanism for Dynamic VM Provisioning and Allocation in Clouds

Sharrukh Zaman, *Student Member*, *IEEE*, and Daniel Grosu, *Senior Member*, *IEEE*

Abstract—Cloud computing providers provision their resources into different types of virtual machine (VM) instances that are then allocated to the users for specific periods of time. The allocation of VM instances to users is usually determined through fixed-price allocation mechanisms that cannot guarantee an economically efficient allocation and the maximization of cloud provider's revenue. A better alternative would be to use combinatorial auction-based resource allocation mechanisms. This argument is supported by the economic theory; when the auction costs are low, as is the case in the context of cloud computing, auctions are especially efficient over the fixed-price markets because products are matched to customers having the highest valuation. The existing combinatorial auction-based VM allocation mechanisms do not take into account the user's demand when making provisioning decisions, that is, they assume that the VM instances are statically provisioned. We design an auction-based mechanism for dynamic VM provisioning and allocation that takes into account the user demand, when making provisioning decisions. We prove that our mechanism is truthful (i.e., a user maximizes its utility only by bidding its true valuation for the requested bundle of VMs). We evaluate the proposed mechanism by performing extensive simulation experiments using real workload traces. The experiments show that the proposed mechanism yields higher revenue for the cloud provider and improves the utilization of cloud resources.

Index Terms—Cloud computing, VM allocation, VM provisioning, dynamic resource configuration, combinatorial auctions

1 Introduction

Cloud providers provision their resources into virtual machine (VM) instances and allocate them to the users for specific periods of time. Provisioning, allocating, and pricing these VM instances are challenging issues that have to be addressed by the cloud providers. The fixed-price allocation mechanisms employed by commercial cloud providers (e.g., Microsoft Azure [1], Amazon EC2 [2]) cannot allocate the VM instances efficiently or price the resources reflecting the dynamically changing user demands. Economic theory suggests that when the auction costs are low, auctions are efficient over the fixed-price mechanisms because products are matched to customers having the highest valuation [3]. In particular, combinatorial auction-based mechanisms are best suited for resource allocation in clouds because of the nature of the allocation requests. However, we have to overcome certain challenges while using combinatorial auction-based mechanisms for VM provisioning and allocation in clouds. The winner determination in a combinatorial auction is an NP-hard problem [4], and therefore, solving it for large number of users and resources will require considerable amount of time. Since the majority of current cloud providers serve a large number of users and have a large amount of resources available for allocation they will need to employ approximation algorithms to solve the winner determination problem in a reasonable amount of time.

In our previous work [5], we designed two combinatorial auction-based approximation mechanisms for VM instance allocation. Although these mechanisms are able to increase the allocation efficiency of VM instances and also increase the cloud provider's revenue, they assume static provisioning of VM instances. That is, they require that the VM instances are already provisioned and would not change. Static provisioning leads to inefficiencies due to underutilization of resources if the mechanism cannot accurately predict the user demand. Since a regular auction computes the price of the items based on user demands, a very low demand may require the auctioneer to set a reserve price to prevent losses.

In this paper, we address the VM provisioning and allocation problem by designing a combinatorial auction-based mechanism that produces an efficient allocation of resources and high profits for the cloud provider. The mechanism extends one of the mechanisms we proposed in [5] to include dynamic configuration of VM instances and reserve prices. The proposed mechanism, called CA-PROVISION, treats the set of available computing resources as "liquid" resources that can be configured into different numbers and types of VM instances depending on the requests of the users. Each user desires a specific bundle of VM instances and bids only on one such bundle (i.e., the users are *single minded*). The mechanism determines the allocation based on the users' valuations until all resources are allocated. It involves a reserve price determined by the operating cost of the resources. The reserve price ensures

• *The authors are with the Department of Computer Science, Wayne State University, 5057 Woodward Ave., Suite 14001.4, Detroit, MI 48202. E-mail: {sharrukh, dgrosu}@wayne.edu.*

Manuscript received 19 Feb. 2013; revised 20 Aug. 2013; accepted 20 Sept. 2013; published online 30 Sept. 2013.
Recommended for acceptance by O. Rana.
For information on obtaining reprints of this article, please send e-mail to: tcc@computer.org, and reference IEEECS Log Number TCC-2013-02-0032.
Digital Object Identifier no. 10.1109/TCC.2013.9.

that a user pays a minimum amount to the cloud provider so that the provider does not suffer any losses from the VM provisioning and allocation.

1.1 Our Contribution

Our main contribution is the design of a combinatorial auction-based mechanism for dynamic provisioning and allocation of VM instances in clouds. The existing combinatorial auction-based VM allocation mechanisms do not take into account the user's demand when making provisioning decisions, that is, they assume that the VM instances are statically provisioned. Our design is novel in the sense that it eliminates the static provisioning requirement and takes into account the changing user demand when making allocation decisions. Our design also includes reserve prices and a method for determining them. We prove that the proposed mechanism is truthful, that is, it guarantees that a participating user maximizes its utility only by bidding its true valuation for the bundle of VMs. We evaluate our mechanism by performing extensive simulation experiments using traces of real workloads from the Parallel Workloads Archive [6]. These extensive experiments show that the proposed mechanism yields higher revenue for the cloud provider and improves the utilization of cloud resources. Finally, we analyze the drawbacks and benefits of employing the proposed mechanism and provide guidelines for its implementation.

1.2 Organization

The rest of the paper is organized as follows: In Section 2, we formulate the problem of dynamic VM provisioning and allocation in clouds. In Section 3, we discuss the related work. In Section 4, we present our proposed mechanism for solving the VM provisioning and allocation problem and characterize its theoretical properties. In Section 5, we perform extensive simulations using real workload traces to investigate the properties of our proposed mechanism. In Section 6, we conclude the paper and discuss possible future research directions.

2 Dynamic VM Provisioning and Allocation Problem (DVMPA)

Virtualization technology allows the cloud computing providers to configure computational resources into virtually any combination of different types of VMs. Hence, it is possible to determine the best combination of VM instances through a combinatorial auction and then dynamically provision them. This will ensure that the number of VM instances of different types are determined based on the market demand and then allocated efficiently to the users. We formulate the *DVMPA* as follows.

A cloud provider offers computing services to users through m different types of VM instances, $VM_1, \ldots, VM_m$. The computing power of a VM instance of type VM_i, $i = 1, \ldots, m$ is w_i, where $w_1 = 1$ and $w_1 < w_2 < \cdots < w_m$. We denote by $\mathbf{w} = (w_1, w_2, \ldots, w_m)$ the vector of computing powers of the m types of VM instances. In the rest of the paper, we will refer to this vector as the "weight vector." As an example of how we use this vector, let us consider a cloud provider offering three types of VM instances: VM_1, consisting of one 2-GHz processor, 4-GB memory, and 500-GB storage; VM_2, consisting of two 2-GHz processors, 8-GB memory, and 1-TB storage; and VM_3, consisting of four 2-GHz processors, 16-GB memory, and 2-TB storage. The weight vector characterizing the three types of VM instances is, thus, $\mathbf{w} = (1, 2, 4)$. We consider the weight-based model for VM instances to make the bidding and allocation more practical. First, considering the CPU, memory, and storage separately for composing the user's bundles will make the users' task of bidding very complex. Complex bidding will be a deterrent to many of the users. On the other hand, the current cloud providers are bundling different combinations of resources into different types of VM instances (e.g., Amazon EC2's High-Memory, High-CPU instances [7]) that the users could choose based on their specific needs. Since the provider knows the computational power of its VM types, all she needs to do is calculate a weight factor for all types of VMs and consider them as components of the weight vector $\mathbf{w}$. Following the practice of current cloud providers, such as Amazon EC2, we consider that the weight is determined mainly by the number of cores. Since the current cloud providers allocate matching resources for memory and storage, we also consider this in our model.

We assume that the cloud provider has enough resources to create a maximum of M VM instances of the least powerful type, VM_1. The cloud provider can provision the VM instances in several ways according to the specified types given by $VM_1, \ldots, VM_m$. Let us denote by k_i the number of VM_i instances provisioned by the cloud provider. The provider can provision any combination of instances given by the vector $(k_1, k_2, \ldots, k_m)$ as long as

$$\sum_{i=1}^{m} w_i k_i \leq M. \tag{1}$$

We consider n users $u_1, \ldots, u_n$ who request computing resources from the cloud provider specified as bundles of VM instances. A user u_j requests VM instances by submitting a bid $B_j = (r_1^j, \ldots, r_m^j, v_j)$ to the cloud provider, where r_i^j is the number of instances of type VM_i requested and v_j is the price user u_j is willing to pay to use the requested bundle of VMs for a unit of time. An example of a bid submitted by a user to a cloud provider that offers three types of VMs can be $B_j = (2, 1, 4, 10)$. This means that the user is bidding 10 units of currency for using two instances of type VM_1, one instance of type VM_2, and four instances of type VM_3 for one unit of time. Here, we assume that the users are *single minded*, i.e., a user bids for only one bundle. Based on the auction outcome, the cloud provider will either allocate the entire bundle to the user or not provide any VM instance at all. The provider runs a mechanism, in our case an auction, periodically (e.g., once an hour) to provision and allocate the VM instances such that its profit is maximized. Thus, the users bid for obtaining the requested VM bundles for one unit of time. If the user's job requires a bundle for more than one unit of time, the user has to bid again in the next round of mechanism execution. To define the profit obtained by the cloud provider, we need to introduce additional notation. Let us denote by p_j the amount paid by user u_j for using her requested bundle of VMs. Note that

depending on the pricing and allocation mechanism used by the cloud provider, p_j and v_j can have different values, usually $p_j < v_j$.

Let us assume that the time interval between two consecutive auctions is one unit of time. Let c_R and c_I be the costs associated with running, respectively, idling a VM_1 instance for one unit of time. Obviously, $c_R > c_I$. The cloud provider's cost of running all available resources (i.e., all M VM_1 instances) is $M \cdot c_R$, while the cost of keeping all the available resources idle is $M \cdot c_I$. We denote by $x = (x_1, x_2, \ldots, x_n)$ the allocation vector, where $x_j = 1$ if the bundle $(r_1^j, \ldots, r_m^j)$ requested by user u_j is allocated to her, and $x_j = 0$, otherwise. Given a particular allocation vector and payments, the cloud provider's profit is given by

$$\Pi = \sum_{j=1}^{n} x_j p_j - c_R \sum_{j=1}^{n} x_j s_j - c_I \left(M - \sum_{j=1}^{n} x_j s_j \right), \quad (2)$$

where $s_j = \sum_{i=1}^{m} w_i r_i^j$ is the amount of "unit" computing resources requested by user u_j. The "unit" computing resource is equivalent to one VM instance of type VM_1 (i.e., the least powerful instance offered). The first term of the equation gives the revenue, the second term gives the running cost of the VM instances that are allocated to the users, and the third term gives the cost of keeping the remaining resources idle.

The purpose of considering the costs c_R and c_I is to reduce the cloud provider's losses when the demand is very low or when the user valuations are too low. In both cases, allocating the resources to the users via an auction will lead to very low user payments, which will implicitly result in loss of revenue for the auctioneer. As we will see later in the paper, considering these costs helps determine a "reserve price" that will prevent users with very low bids from participating in the auction and implicitly guarantees that the provider is still making some profit when the demand is low.

The *DVMPA* problem is defined as follows:

$$\max \Pi, \quad (3)$$

subject to

$$\sum_{j=1}^{n} x_j s_j \leq M, \quad (4)$$

$$x_j \in \{0, 1\}, \quad j = 1, \ldots, n, \quad (5)$$

$$0 \leq p_j \leq v_j, \quad j = 1, \ldots, n. \quad (6)$$

The solution to this problem consists of allocation x_j and price p_j for each user u_j who requested the bundle $(r_1^j, \ldots, r_m^j)$, $j = 1, \ldots, n$. The allocation will determine the number of VMs of each type that needs to be provisioned as follows. We compute $k_i = \sum_{j=1}^{n} x_j r_i^j$, for each type VM_i and provision k_i VM instances of type VM_i.

Current cloud service providers use a fixed-price mechanism to allocate the VM instances and rely on statistical data to provision the VMs in a static manner. In our previous work [5], we have shown that combinatorial auction-based mechanisms can allocate VM instances in clouds generating higher revenue than the currently used fixed-price mechanisms. However, the combinatorial auction-based mechanisms we explored in our previous work [5] require that the VMs are provisioned in advance, that is, they require static provisioning. We argue that the overall performance of the system can be increased by carefully selecting the set of VM instances in a dynamic fashion, which reflects the market demand at the time when an auction is executed. In Section 4, we propose a combinatorial auction-based mechanism that solves the DVMPA problem by determining the allocation, pricing, and the best configuration of VMs that need to be provisioned by the cloud provider to obtain higher profits. Since very little is known about profit maximizing combinatorial auctions [4], we cannot provide theoretical guarantees that our auction-based mechanism maximizes the profit. The only guarantee we can provide is that the mechanism determines an approximately *efficient allocation* (i.e., approximately maximizes the sum of the users' valuations). In designing our mechanism, we also use reserve prices that are known to increase the revenue of the auctioneer, in our case, the revenue of the cloud provider.

3 Related Work

Researchers approached the problem of VM provisioning in clouds from different points of view. Shivam et al. [8] presented two systems called Shirako and NIMO that complement each other to obtain on-demand provisioning of VMs for database applications. Shirako does the actual provisioning and NIMO guides it through active learning models. The CA-PROVISION mechanism we propose in this paper performs both demand tracking and provisioning via a combinatorial auction. Dynamic provisioning of computing resources was investigated by Quiroz et al. [9] who proposed a decentralized online clustering algorithm for VM provisioning based on the workload characteristics. The authors proposed a model-based approach to generate workload estimates on a long-term basis. Our proposed mechanism provisions the VMs dynamically and it does not require the prediction of the workload characteristics, rather the current demand for VMs is captured and the provisioning is decided by a combinatorial auction-based mechanism. Vecchiola et al. [10] proposed a deadline-driven provisioning mechanism supporting the execution of scientific applications in clouds.

Recently, researchers investigated economic models for resource allocation in computational grids. Wolski et al. [11] compared commodities markets and auctions in grids in terms of price stability and market equilibrium. Das and Grosu [12] proposed a combinatorial auction-based protocol for resource allocation in grids. They considered a model where different grid providers can provide different types of computing resources. An "external auctioneer" collects the information about resources and runs a combinatorial auction-based allocation mechanism, where users participate by requesting bundles of resources.

Several researchers have investigated the economic aspects of cloud computing from different points of view. Wang et al. [13] studied different economic and system implications of pricing resources in clouds. Altmann et al. [14] proposed a marketplace for resources, where the

allocation and pricing are determined using an exchange market of computing resources. In this exchange, the service providers and the users both express their ask and bid prices and matching pairs are granted the allocation and removed from the system. Risch et al. [15] designed a testbed for cloud services that enables the testing of different mechanisms on clouds. They deployed the exchange mechanism described by Altmann et al. [14] on this platform. In this paper, we consider designing a combinatorial auction mechanism with reserve price instead of an exchange. In this case, instead of specifying an asking price, the cloud provider determines a reserve price that is based on its cost parameters. Also, the outcome of the auction determines the configuration of VM instances that needs to be provisioned.

Amazon EC2 Spot Instances (SI) [16] is an example of auction-based mechanism used in a commercial cloud. Although Amazon publishes historic spot prices, the method used for determining the prices is not disclosed [17]. Therefore, most research work about the Spot Instances revolves around the issue of how SI can be utilized to reduce the costs of running different applications on Amazon EC2. Wee [18] showed that the SIs are about 52 percent cheaper and that shifting the time of computation in SIs can save about 4 percent of the cost. Zhang et al. [19] addressed the problem of dynamically allocating computing resources for different VM types in Amazon Spot Instances to maximize the revenue. Ben-Yehuda et al. [17] showed that the SI prices are not necessarily market driven.

The complexity of solving combinatorial auctions, specifically the winner determination problem, was first addressed by Rothkopf et al. [20]. Sandholm [21] proved that solving the winner determination problem is computationally hard. Zurel and Nisan [22] proposed an approximation algorithm for solving combinatorial auctions. The book by Cramton et al. [23] provides good foundational knowledge on combinatorial auctions. Lehmann et al. [24] initiated the study of combinatorial auctions with single-minded bidders and devised a greedy mechanism for combinatorial auctions. Mu'alem and Nisan [25] showed how to obtain truthful mechanisms for single-minded settings by combining approximation algorithms. Briest et al. [26] and Chekuri and Gamzu [27] proposed additional construction techniques for the design of truthful mechanisms in single-minded settings.

In our previous work [5], we extended the mechanism proposed by Lehmann et al. [24] and developed CA-GREEDY, a combinatorial auction-based mechanism to allocate VM instances in clouds. We showed that CA-GREEDY obtains an allocation of VM instances that is approximately efficient and generates higher revenue than the currently used fixed-price mechanisms. However, CA-GREEDY requires that the VMs are provisioned in advance, that is, it requires static provisioning. The mechanism we propose in this paper is different from CA-GREEDY in that it selects the set of VM instances in a dynamic fashion, which reflects the market demand at the time when the mechanism is executed.

Several researchers addressed the design of auction mechanisms for resource allocation in clouds, the closest works to ours being by Lampe et al. [28] and Prasad et al. [29]. Lampe et al. [28] proposed an equilibrium price auction mechanism considering the physical machine capacities in determining the number of VMs to be provisioned. Their proposed mechanism provides an approximate solution to the procurement auction, but in contrast to our work, it does not guarantee truthfulness. Prasad et al. [29] formulated the resource allocation problem as a procurement auction. In their model, the users express their requirements to an auction broker and cloud providers participate in an auction run by the broker agent. This approach assumes the existence of several cloud providers with the auction taking place among them. This is different from our setting that focuses on auction mechanisms run by individual cloud providers. Another important difference from our work is that their mechanism is not truthful.

4 Combinatorial Auction-Based Dynamic VM Provisioning and Allocation Mechanism

We present a combinatorial auction-based mechanism, called CA-PROVISION, that computes an approximate solution to the DVMPA problem. That is, it determines the prices the winning users have to pay, and the set of VM instances that need to be provisioned to meet the winning users' demand. The mechanism also ensures that the maximum possible number of resources is allocated and no VM instance is allocated for less than the reserve price. The design of the mechanism is based on the ideas presented in [24].

CA-PROVISION uses a reserve price to guarantee that users pay at least a given amount determined by the cloud provider. Thus, the cloud provider needs to set the reserve price, denoted by v_{res}, to a value that depends on its costs associated with running the VMs. To do that, we observe that the reserve price should be the break-even point between c_R and c_I, which is given by $c_R - c_I$. This is because if a unit resource is not allocated, it incurs a loss of c_I. Again, if this resource is allocated for a price $c_R - c_I$, the loss is $c_R - (c_R - c_I) = c_I$. In other words, the minimum price a user has to pay for using the least powerful VM for a unit of time is equal to the difference between the cost of running and the cost of keeping the resource idle. An auction with reserve price v_{res} can be modeled by an auction without reserve price in which we artificially introduce a dummy bidder u_0 having as its valuation the reserve price, i.e., $v_0 = v_{res}$. The dummy user u_0 bids $B_0 = (1, 0, \ldots, 0, v_{res})$, i.e., $r_1^0 = 1$, $r_i^0 = 0$ for all $i = 2, \ldots, m$, and $v_0 = v_{res}$. CA-PROVISION uses the density of the bids to determine the allocation. User u_j's *bid density* is $d_j = v_j/s_j$, where $s_j = \sum_{i=1}^{m} w_i r_i^j$, $j = 0, \ldots, n$. The bid density is a measure of how much a user bids per unit of allocation. In our case, the unit of allocation corresponds to one VM instance of type VM_1. To guarantee that the users are paying at least the reserve price, the mechanism will discard all users for which $d_j < d_0$.

CA-PROVISION first collects bids from users, calculates the bid density for all bids, and sorts the bids according to their bid density. It then calculates the reserve price and discards bids whose bid density falls below the reserve price. Next, it allocates computing resources to the users in the sorted order and provisions the resources accordingly.

Finally, it calculates the payment of each winning user, that is, the amount they must pay to the cloud provider. The payment is the minimum value a winning user must bid to obtain the resources she requests (i.e., the critical payment [24]). All losing users pay zero.

CA-PROVISION is given in Algorithm 1. The mechanism requires some information from the system such as the total amount of computing resources M, expressed as the total number of VMs of type VM_1 that can be provisioned by the cloud provider. The mechanism also requires as input the number of available VM types, m, and their weight vector $\mathbf{w}$. It also needs to know c_R, the cost of running a VM instance of type VM_1, and c_I, the cost of keeping idle a VM instance of type VM_1.

Algorithm 1. CA-PROVISION Mechanism.
Require: M; m; $w_j : j = 1, \ldots, n$; c_R; c_I;
Ensure: W; $p_j : j = 1, \ldots, n$; $k_i : i = 1, \ldots, m$;
1: {*Phase 1: Collect bids*}
2: **for** $j = 1, \ldots, n$ **do**
3: collect bid $B_j = (r_1^j, \ldots, r_m^j, v_j)$ from user u_j
4: **end for**
5: {*Phase 2: Winner determination and provisioning*}
6: $W \leftarrow \emptyset$ {set of winners}
7: $v_{res} \leftarrow c_R - c_I$
8: add dummy user u_0 with bid $B_0 = (1, 0, 0, \ldots, 0, v_{res})$
9: **for** $j = 0, \ldots, n$ **do**
10: $s_j \leftarrow \sum_{i=1}^{m} r_i^j w_i$
11: $d_j \leftarrow v_j / s_j$ {'bid density'}
12: **end for**
13: re-order users $u_1, \ldots, u_n$ such that $d_1 \geq d_2 \geq \cdots \geq d_n$
14: let l be the index such that $d_j \geq d_0$ if $j \leq l$, and $d_j < d_0$ otherwise
15: discard users $u_{l+1}, \ldots, u_n$
16: rename user u_0 as u_{l+1}
17: set $n \leftarrow l + 1$
18: $R \leftarrow M$
19: **for** $j = 1, \ldots, n-1$ **do** {leave out dummy user}
20: **if** $s_j \leq R$ **then**
21: $W \leftarrow W \cup u_j$
22: $R \leftarrow R - s_j$
23: **end if**
24: **end for**
25: **for** $i = 1, \ldots, m$ **do** {determine VM configuration}
26: $k_i \leftarrow \sum_{j:u_j \in W} r_i^j$
27: **end for**
28: {*Phase 3: Payment*}
29: **for all** $u_j \in W$ **do**
30: $W'_j \leftarrow \{u_l : u_l \notin W \wedge (v_j = 0 \Rightarrow u_l \in W)\}$
31: $l \leftarrow$ lowest index in W'_j
32: $p_j \leftarrow d_l s_j$
33: **end for**
34: **for all** $u_j \notin W$ **do**
35: $p_j \leftarrow 0$
36: **end for**
37: **return** $(W; p_j : j = 1, \ldots, n; k_i : i = 1, \ldots, m)$

The mechanism works in three phases. In Phase 1, it collects the users' bids B_j (lines 1 to 4). In Phase 2, the mechanism determines the winning bidders and the VM configuration that needs to be provisioned by the cloud provider as follows: It adds a dummy user u_0 with a bid that contains only one instance of VM_1 and has a valuation of $v_{res} = c_R - c_I$ (line 8). This dummy user is only used to model the auction with reserve price and will not receive any allocation. It then computes the bundle size s_j and bid density d_j of all users (lines 9 to 12). Then, all users except the dummy user are ordered in decreasing order of their bid densities and all users u_j with $d_j < d_0$ are discarded (lines 13 to 15). The dummy user u_0 is then moved to the end of the list of the remaining users because it has the lowest density in the current set of users. The mechanism reassigns n to be the total number of users under consideration, including the dummy user (lines 16 and 17).

Next, the mechanism determines the winning users in a greedy fashion. It allocates the requested bundles to users in decreasing order of their bid density, as long as there are resources available (lines 18 to 24). However, the dummy user is not considered for allocation. Once the winners are determined, the mechanism determines the VM configuration that needs to be provisioned by aggregating the bundles requested by the winning users (lines 25 to 27).

In Phase 3, the mechanism determines the payment for all users. For each winning bidder u_j, the mechanism finds the set of losing bidders W'_j who would otherwise win if $v_j = 0$, i.e., when user u_j is not participating (line 30). From this set, user u_l with the highest bid density is selected. This is determined by taking the lowest indexed user from set W'_j, since the set of users is already sorted in nondecreasing order of users' bid densities (line 31). User u_j's payment is then calculated by multiplying her bundle size s_j with the bid density d_l of u_l. All losing bidders pay zero. This type of payment is known in the mechanism design literature as the *critical payment* [24]. The reason we choose this type of payment is that it is a necessary condition for obtaining a *truthful mechanism*, (i.e., a mechanism that provides incentives to the users to bid their true valuations for the requested bundles). In the next section, we show that our proposed mechanism is truthful.

4.1 Properties of CA-PROVISION

We now investigate the properties of the proposed mechanism. An important property of a mechanism is *incentive compatibility*, which is also called *truthfulness*. This is important because the mechanism computes the allocation and payment based on the information reported by the users (i.e., bids), which is private information. A rational user may manipulate the mechanism by bidding false valuations if it benefits her to do so. The challenge of designing a mechanism, therefore, involves designing the winner determination and payment functions that give the users incentives to bid truthfully. This is very important because the users participating in a truthful allocation mechanism do not have to employ sophisticated bidding strategies to maximize their utilities. They just need to bid their true valuation for the bundle of VMs.

In the following, we denote by $B = (B_1, \ldots, B_n)$, the vector representing the bids of all users and, by $B_{-j} = (B_1, \ldots, B_{j-1}, B_{j+1}, \ldots, B_n)$ the vector of all user's bids

except the bid B_j of user u_j. Hence, B can also be represented as $B = (B_j, B_{-j})$. We also assume that $B_j = (r_1^j, \ldots, r_m^j, v_j)$ is the "true bid" of the user, i.e., the user requires the bundle $(r_1^j, \ldots, r_m^j)$ and she values it at v_j. We denote by $\hat{B}_j = (\hat{r}_1^j, \ldots, \hat{r}_m^j, \hat{v}_j)$, the bid the user submits to the mechanism, which may or may not be the same as B_j. We denote by $\hat{B} = (\hat{B}_1, \ldots, \hat{B}_n)$ the vector of all user's bids reported to the mechanism.

Here, we also abuse the notations for the set of winners W and the payments $p_1, \ldots, p_n$. We will use them as the winner determination function $W(.)$ and the payment functions $p_1(.), \ldots, p_n(.)$. $W(\hat{B})$ computes the set of winners from the bid vector $\hat{B}$ and $p_j(\hat{B})$ computes the payment for user u_j from $\hat{B}$. We express the fact that user u_j values her requested bundle at v_j by the *valuation function*:

$$V_j(W(\hat{B}), B_j) = \begin{cases} v_j, & \text{if } u_j \in W(\hat{B}), \\ 0, & \text{otherwise.} \end{cases} \quad (7)$$

That is, user u_j obtains a valuation of v_j if her requested bundle is allocated and a valuation of 0, otherwise.

The utility that user u_j derives from obtaining the requested bundle is the difference between her valuation V_j and payment p_j (i.e., "quasi-linear" utility) as follows:

$$U_j(W(\hat{B}), B_j) = V_j(W(\hat{B}), B_j) - p_j(\hat{B}). \quad (8)$$

We assume that the users are rational, that is, their goals are to maximize their utilities. A truthful mechanism guarantees that a user maximizes her utility only by bidding her true valuation for the bundle. In the following, we define the concept of a truthful mechanism.

Definition 1 (Truthful mechanism). *A mechanism defined by the winner determination function $W(.)$ and payment functions $p_1(.), \ldots, p_n(.)$ is* truthful *if for all u_j, $\hat{B}_j$, and $\hat{B}_{-j}$,*

$$U_j(W(B_j, \hat{B}_{-j}), B_j) \geq U_j(W(\hat{B}_j, \hat{B}_{-j}), B_j). \quad (9)$$

That is, a user participating in a truthful mechanism maximizes her utility only by bidding her true valuation for the bundle regardless of the other users' bids.

Truthfulness was well investigated and characterized in the mechanism design literature [4]. One such useful characterization gives the conditions under which a mechanism is truthful. Stated informally, a mechanism is truthful if the allocation function is *monotone* and the payments are the *critical payments* [25]. We define these properties in the context of CA-PROVISION below.

Definition 2 (Monotonicity). *An allocation function $W(.)$ is* monotone *if for every user u_j and every $\hat{B}_{-j}$, $B_j = (r_1^j, \ldots, r_m^j, v_j)$ is a winning bid, then every $B'_j = (r'^j_1, \ldots, r'^j_m, v'_j)$ with $s'_j \leq s_j$ and $v'_j \geq v_j$ is also a winning bid. Here, $s'_j = \sum_{i=1}^m w_i r'^j_i$ and $s_j = \sum_{i=1}^m w_i r_i^j$.*

In other words, an allocation function is monotone if a winning user also wins if she bids a higher valuation for a smaller size bundle.

Definition 3 (Critical value). *The critical value v_j^c for user $u_j \in W(\hat{B})$ is defined as the unique value such that $B_j = (\hat{r}_1^j, \ldots, \hat{r}_m^j, v_j)$ is a winning bid for any $v_j \geq v_j^c$ and a losing bid for any $v_j \leq v_j^c$.*

Thus, the critical value is the minimum valuation a user must declare to obtain her requested bundle.

Definition 4 (Critical payment). *The critical value payment function p associated with monotone allocation function $W(\hat{B})$ is defined by: $p_j = v_j^c$, if user u_j wins, and $p_j = 0$, otherwise.*

In other words, under the critical payment function a winning user pays her critical value, while a losing user pays zero (i.e., the mechanism is a normalized mechanism).

Next, we present two lemmas and one theorem to prove that CA-PROVISION is truthful.

Lemma 1. *CA-PROVISION implements a monotone allocation function.*

Proof. CA-PROVISION allocates resources to users in nonincreasing order of $d_j = v_j/s_j$, where s_j is the sum of the weights of VMs in the requested bundle. Hence, a bid with higher v_j and lower s_j is preferable to the mechanism. Assume user u_j gets the allocation by bidding $B_j = (r_1^j, \ldots, r_m^j, v_j)$. If she changes her bid to $\hat{B}_j = (r_1^j, \ldots, r_m^j, \hat{v}_j)$, where $\hat{v}_j \geq v_j$, she stays at least at the same rank in the ordered list. Since she is requesting the same resource, this implies that her bid is a winning bid. On the other hand, if user u_j bids $\hat{B} = (\hat{r}_1^j, \ldots, \hat{r}_m^j, v_j)$, where $\hat{s}_j = \sum_{i=1}^m w_i \hat{r}_i^j \leq s_j$, then d_j increases and user u_j stays at least at the same rank in the greedy order of users (Algorithm 1, line 13). Since she is requesting fewer resources, her bid $\hat{B}_j$ is a winning bid. By Definition 2, CA-PROVISION implements a monotone allocation function. □

Lemma 2. *CA-PROVISION charges the winning users their critical payments.*

Proof. To compute the payment for a winning user u_j, CA-PROVISION finds a losing user u_l who would win if user u_j would not participate. That means user u_j needs to defeat user u_l with her bid to get her required bundle (i.e., $d_j \geq d_l$). This means that $v_j/s_j \geq d_l$, and therefore $v_j \geq d_l \cdot s_j$. CA-PROVISION charges $p_j = d_l \cdot s_j$ to user u_j (line 32 of Algorithm 1) which is the minimum valuation u_j must bid to obtain her required bundle. The losing users pay zero (Lines 34 and 35 of Algorithm 1). Therefore, CA-PROVISION implements the critical value payment (Definition 4). □

Theorem 1. *CA-PROVISION is truthful.*

Proof. According to Lemmas 1 and 2, CA-PROVISION implements a monotone allocation function and charges the winning users their critical payments. Following the results of Mu'alem and Nisan [25], CA-PROVISION is a truthful mechanism. The reserve prices do not affect the truthfulness of the mechanism because they are basically bids put out by the dummy user controlled by the cloud provider and truthful bidding is still a dominant strategy for the users. □

Now, we investigate the complexity of CA-PROVISION. The loops in lines 19-24 and lines 29-33 constitute the major computational load of Algorithm 1. The first loop has a worst-case complexity of $O(M)$. The worst case is when all winning bidders bid for bundles containing exactly one unit

of VM_1 instances. The total execution time of the loop in lines 29-33 is $O(n)$. This is because it iterates over the set of winning bidders and the search is performed on the losing bidders. Since the bidders are already sorted, the search for a critical payment for a winner u_{j+1} actually starts from the "critical payment bidder" u_l of u_j (without loss of generality, we assume both u_j and u_{j+1} are winners in this case). Hence, the overall worst-case complexity of this loop is $O(n)$, whereas the sorting in line 13 costs $O(n \log n)$. Thus, the complexity of CA-PROVISION is $O(M + n \log n)$.

5 Experimental Results

We perform extensive simulation experiments with real workload data to evaluate the CA-PROVISION mechanism. We compare the performance of CA-PROVISION with the performance of a combinatorial auction-based mechanism, called CA-GREEDY, that uses static VM provisioning [5]. In our previous work [5], we investigated the performance of CA-GREEDY against the performance of the fixed-price VM allocation mechanism in use by current cloud providers. The mechanism showed significant improvements over the fixed-price allocation mechanism, thus, making it a good candidate for our current experiments.

CA-GREEDY mechanism uses the same type of payment determination as CA-PROVISION. The difference between the two is in how they allocate the VM instances. The CA-GREEDY assumes that the VM instances are already provisioned (i.e., static provisioning), while CA-PROVISION makes the provisioning decisions dynamically.

We perform a total of 264 experiments with data generated using 11 workload logs from the Parallel Workloads Archive [6] and 24 different combination of other parameters for each workload. In this section, we describe the experimental setup and discuss the experimental results.

5.1 Experimental Setup

The experiments consist of generating job submissions from a given workload and then running both CA-GREEDY and CA-PROVISION concurrently to allocate the jobs and provision the VMs. For setting up the experiments, we have to address several issues such as workload selection, bid generation, and setting up the auction. We discuss all these issues in the following sections.

5.1.1 Workload Selection

To the best of our knowledge, standard cloud computing workloads were not publicly available at the time of writing this paper. Thus, to overcome this limitation, we rely on well studied and standardized workloads from the Parallel Workloads Archive [6]. This archive contains a rich collection of workloads from various grid and supercomputing sites. Out of the 26 real workloads available, we selected 11 logs that were recorded most recently. These logs are:

1. ANL-Intrepid-2009, from a Blue Gene/P system at Argonne National Lab;
2. DAS2-fs0-2003-DAS-fs4-2003, from a research grid of five clusters at the Advanced School of Computing and Imaging in the Netherlands;
3. LLNL-Atlas-2006 and LLNL-Thunder-2007 from two Linux clusters (Atlas and Thunder) located at Lawrence Livermore National Lab;
4. LLNL-uBGL-2006, from a Blue Gene/L system at Lawrence Livermore National Lab;
5. LPC-EGEE-2004, from a Linux cluster at The Laboratory for Corpuscular Physics, University Blaisé-Pascal, France; and
6. SDSC-DS-2004, from a 184-node IBM eServer pSeries 655/690 called DataStar located at the San Diego Supercomputer Center.

TABLE 1
Workload Logs

Logfile	Duration	Jobs	Processors
ANL-Intrepid-2009	8 months	68,936	163,840
DAS2-fs0-2003	12 months	225,711	144
DAS2-fs1-2003	12 months	40,315	64
DAS2-fs2-2003	12 months	66,429	64
DAS2-fs3-2003	12 months	66,737	64
DAS2-fs4-2003	11 months	33,795	64
LLNL-Atlas-2006	8 months	42,725	9,216
LLNL-Thunder-2007	5 months	121,039	4,008
LLNL-uBGL-2006	7 months	112,611	2,048
LPC-EGEE-2004	9 months	234,889	140
SDSC-DS-2004	13 months	96,089	1,664

In Table 1, we provide a brief description of the workloads we use in our experiments. The table contains the name of the log file, the length of time the logs were recorded, the total number of submitted jobs, and the total number of processors available in the system. The log file name generally contains the acronym of the organization, the name of the system, and the year of its generation. From the duration column, we see that the logs were generated for long periods of time, as long as 13 months for the SDSC log. The number of jobs submitted ranges from many thousands to more than a couple of hundred thousands, while the number of processors ranges from 64 to 163,840. These large variations in the number of processors and the number of submitted jobs make these logs very suitable for experimentation, providing us with a wide range of simulation scenarios.

The workloads are given in the standard workload format (swf) described in [30]. In this format, the information corresponding to every job submitted to the system is stored as a record with 18 fields. To generate the workload for our simulation experiments, we need the information from six fields of the log files as follows:

1. *Job number*: stores the job's identifier.
2. *Submit time*: stores the job submission time.
3. *Runtime*: stores the time the job needs to complete its execution. We use this as the time required to complete the job. We round this up to the nearest hour because we run hourly auctions in the experiments.
4. *Number of allocated processors*: for our purposes, this represents the number of requested processors.
5. *Average CPU time used*: Average time a CPU was running. We use this field in conjunction with the preceding two parameters to determine the amount of communication and the parallel speedup of the job.
6. *User ID*: stores the ID of the user who submitted the job. We use this ID to place users into different classes having different bidding behaviors.

TABLE 2
Statistics of Workload Logs

Logfile	Duration (hours)	Jobs / hour	Avg. Runtime	Avg procs. per job
ANL-Intrepid-2009	5759	12	2.09	5063
DAS2-fs0-2003	8744	26	1.09	10
DAS2-fs1-2003	8633	5	1.23	8
DAS2-fs2-2003	8760	8	1.29	9
DAS2-fs3-2003	8712	8	1.17	5
DAS2-fs4-2003	7963	4	1.67	4
LLNL-Atlas-2006	4308	10	2.52	401
LLNL-Thunder-2007	3605	34	1.52	43
LLNL-uBGL-2006	5339	21	1.25	576
LPC-EGEE-2004	5728	41	1.80	1
SDSC-DS-2004	9387	10	2.88	62

We list some statistics of the workload files in Table 2.

The logs from the Parallel Workloads Archive [6] were collected from different heterogeneous sources and then converted into the standard format. Therefore, in some logs, some of the fields are not specified because the original files had missing information. Some records in a log file may also have fewer fields than the other records from the same file. We make corrections on these records as follows:

- If the job starting time is missing, we consider it to be equal to the previous job's start time. The logs record the jobs in order of their arrival times. Matching a missing arrival time with the previous job maintains the job order.
- If the execution time is missing, we randomly generate an execution time between 1 and 2 hours from a uniform distribution. As can be seen in Table 2, most of the workloads have an average runtime within this range.
- If the number of processors is missing, we generate a number between 10 and 60 randomly, from a uniform distribution. Since the average number of processors per job differs widely among the workload logs (from 1 processor/job up to 5,063 processors/job), we select a distribution that has a mean (35 processors/job) approximately equal to the average of the two-digit numbers in the list (i.e., 10, 43, and 63 processors/job).
- If the average CPU time is missing, we generate a random number between 50 and 100 percent of the total runtime using the uniform distribution. This generates jobs with communication to computation ratios between 0 and 0.5.
- We assign user IDs randomly in cases in which they are not provided.

5.1.2 Job and Bid Generation

For each record in a log file, we generate a job that a user needs to execute and create a bid for it. There are two important parameters associated with a job that we need to generate, the requested bundle of VMs and the associated bid. First, to generate the bundle of VM instances for a job j, we determine its *communication to computation ratio*, $\rho_j = 1 - \frac{T_j^{CPU}}{T_j^R}$, where T_j^{CPU} is the average CPU time and T_j^R is the total runtime of the job. The communication to computation ratio measures the fraction of the total runtime that is spent by the job on communication and synchronization among its processes. Based on this value, we categorize the job into one of m categories, where m is the number of VM types available. The job category specifies a "first choice" of VM type for the job. This works as follows: We define a factor μ that characterizes how many of the total requested VMs will be requested as "first choice" type VM instances. A job of category i requesting P_j processors will create a bundle comprising a number of VM_i instances required to allocate μP_j processors. The rest of the processors will be requested by arbitrarily choosing other VM types. After creating the bundle, we generate the associated bid. To do that we first determine the speedup of the job, $S_j = P_j \times \frac{T_j^{CPU}}{T_j^R}$, where P_j is the number of CPUs used, T_j^{CPU} is the average CPU time, and T_j^R is the total runtime of the job. This speedup is multiplied by a "valuation factor" to generate the bid. This valuation factor is linked to the type of user. We divide the users into five categories using their user ID, modulo five. The last parameter we set for a job is its deadline. Since there is no deadline information provided in the workload logs, we assume that the deadline is between four and eight times the time required to complete the job. Hence, we set the deadline of a job to the required time multiplied by a random number between 4 and 8. We would like to mention here that the deadline is solely an attribute of the user and that the auction is not aware of the individual deadlines. They are only used to model the user behavior, that is, to determine when the user stops bidding if her requested bundle is not allocated, and whether a user's submitted job is completed or not.

We run CA-GREEDY and CA-PROVISION mechanisms concurrently and independently considering the users who have jobs available for execution. A user (or job) participates in the auction until her job completes or it becomes certain that the job cannot finish by the deadline. A user is "served" if her job completes its execution and "not served," otherwise. Without loss of generality, in the rest of the paper, we assume that each user is submitting only one job and we will use "user" and "job" interchangeably.

5.1.3 Auction Setup

We consider a cloud provider that offers four different types of virtual machine instances, VM_1, VM_2, VM_3, and VM_4. These VM types are characterized by the weight vector $\mathbf{w} = (1, 2, 4, 8)$. From each workload file, we extract N, the total number of users and M, the total number of processors available. The number of users participating in a particular auction is determined dynamically as the auction progresses. That is, n is the number of users that have been generated, not yet been served, and whose job deadlines have not been exceeded yet.

We setup few parameters to generate bundles specific to the jobs submitted by a user. The vector (C_1, C_2, C_3) determines the communication ratios used to categorize the jobs. We use $(C_1, C_2, C_3) = (0.05, 0.15, 0.25)$, as follows: A job having communication ratio below 0.05 is a job of

TABLE 3
Simulation Parameters

Name	Description	Value(s)
N	Total users	From log file
M	Total CPUs	From log file
T	Simulation hours	From log file
(c_I, c_R)	Idle and running cost of unit VM	(.05, .1), (.1, .25), (.15, .5)
μ	Factor for CPUs for 'first choice' VM type	0.5, 0.75
$\mathbf{h}$	Static distribution of processors among VM types	(.25, .25, .25, .25), (.07, .13, .27, .53)
$\mathbf{f}$	Valuation factors for types of users	(.5, 1, 1.5, 2, 2.5), (1, 1.5, 2, 3, 4)
C_1, C_2, C_3	Boundaries of communication ratios	(.05, .15, .25)

type 1 and the majority of its needed VM instances μp_j will be requested as VM_1, where p_j is the number of processors requested by user u_j. We consider the following values for μ, 0.5 and 0.75. The rest of the bundle is arbitrarily determined using the other types of VM instances. We use the user ID field of the log file to determine the valuation range of the user. There are five classes of users submitting jobs. The class t of a user is determined by $((\text{user ID}) \bmod 5)$. The logs have real user IDs, therefore this classification virtually creates a realistic distribution of users. Each class t of users is associated with a "valuation factor" f_t. Having determined that a user is of class t, we determine the valuation of her bundle using the speedup (as shown in the previous section) and the "valuation factor" f_t from the vector $\mathbf{f}$. The vector $\mathbf{f}$ has five elements (equal to the number of classes of users), each representing the mean value of how much a user of that class "values" each "unit of speedup". In particular, a user u_j having a speedup of S_j for her job is willing to pay $f_t S_j$ on average for each hour of her requested bundle of VMs, given that u_j falls in class t. We generate a random value between 0 and $2f_t$, and then multiply it with S_j to generate valuations with a mean of $f_t S_j$. We use two sets of vectors for $\mathbf{f}$, as shown in Table 3.

CA-PROVISION determines by itself the configuration of the VMs that needs to be provisioned by the cloud provider, whereas CA-GREEDY assumes static VM provisioning, and thus, needs the VM configuration provisioned in advance. To generate the static provision of VMs required by CA-GREEDY, we use a vector $\mathbf{h}$ as follows: We use two instances of $\mathbf{h}$ in our simulation experiments. The first one, $\mathbf{h} =$ (0.07, 0.13, 0.27, 0.53) ensures that, given the weight vector $\mathbf{w}$, the number of VM instances of each type is not the same. However, this vector ensures that about the same amount of computing resources as in the case of CA-PROVISION are provisioned. This same amount of resources are then provisioned as different types of VMs in the case of CA-GREEDY. This could be verified by multiplying each component of $\mathbf{h}$ with the corresponding component of $\mathbf{w}$, which gives 0.56, 0.52, 0.54, and 0.53. The other instance of vector $\mathbf{h} = (0.25, 0.25, 0.25, 0.25)$ that we use in our simulations ensures that the total number of processors are equally provisioned into different types of VMs. We also evaluate CA-PROVISION against a modified version of CA-GREEDY, which considers forecasting the demand of each type of VM instance. The forecasted demand is determined by executing CA-PROVISION for the z percentage of the past user requests and calculating the average of the number of VM instances of each type provisioned by CA-PROVISION. This number is then used to statically provision the VM instances when running CA-GREEDY. This types of experiments will give us insight on whether the demand forecast using a moving average can generate comparable performance to that obtained by CA-PROVISION. When running the experiments, we select four values for z: 10, 25, and 50 percent.

We list all simulation parameters in Table 3. With all combinations of values, we perform 24 experiments with each log file. As an implementation note, the frequency of running the mechanisms is to be determined by the system designers based on the performance of the underlying software layer. In our experiments, we executed the mechanisms every hour just to follow the standard practice in Amazon EC2. In a real cloud system, the actual time for provisioning must be considered when determining the frequency of running the mechanisms.

5.2 Analysis of Results

We investigate the performance of the two mechanisms for different workloads. Since the workloads are heterogeneous in several dimensions, we first define a metric to characterize the workloads, and thus, be able to establish an order among them. Then, we normalize the performance metrics of the mechanisms and compare them with respect to the workload characteristics. Finally, we try to gain more insight by comparing the allocation determined by the two mechanisms side by side.

We define a metric for comparing the workload logs as follows: Looking at the workload characteristics listed in Tables 1 and 2, we determine that the best metric to compare the workloads is the *normalized load* η_ω, defined as $\eta_\omega = \frac{J_\omega \times T_\omega \times P_\omega}{M_\omega}$, where η_ω is the normalized load of workload ω, J_ω is the average number of jobs submitted per hour, T_ω is the average runtime of the jobs, P_ω is the average number of processors required per job, and M_ω is the total number of processors in the system corresponding to workload ω. The number of jobs per hour multiplied by the average processors per job determines how many processors are requested by the jobs arriving each hour. Multiplying this with the average runtime gives an estimate of the average number of processors requested by all jobs in an hour. Essentially, the normalized load measures the average amount of load per processor. When analyzing the results, we use the normalized load to rank the otherwise heterogeneous log files.

From each set of simulation experiments, we compute the total revenue generated, the total cost incurred, and the total profit earned by each mechanism. Since the workloads were generated for different durations of time for systems with different number of processors, we scale the profit, revenue, and cost with respect to the total simulation hours and the number of processors. We define the *profit per processor-hour* as $\Pi_\omega^{ph} = \frac{\Pi_\omega}{M_\omega \times L_\omega}$, where Π_ω is the profit computed on workload ω, M_ω is the total number of processors, and L_ω is the number of hours of data provided in workload ω. We define *revenue per processor-hour* and *cost per processor-hour* in a similar fashion.

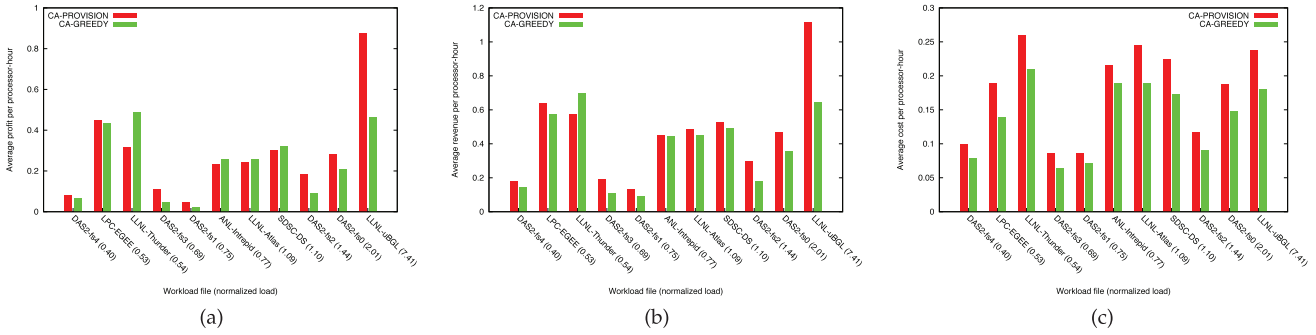


Fig. 1. CA-PROVISION versus CA-GREEDY: (a) Average profit per processor-hour; (b) average revenue per processor-hour; (c) average cost per processor-hour. Horizontal axis shows the log file name with normalized load in parentheses.

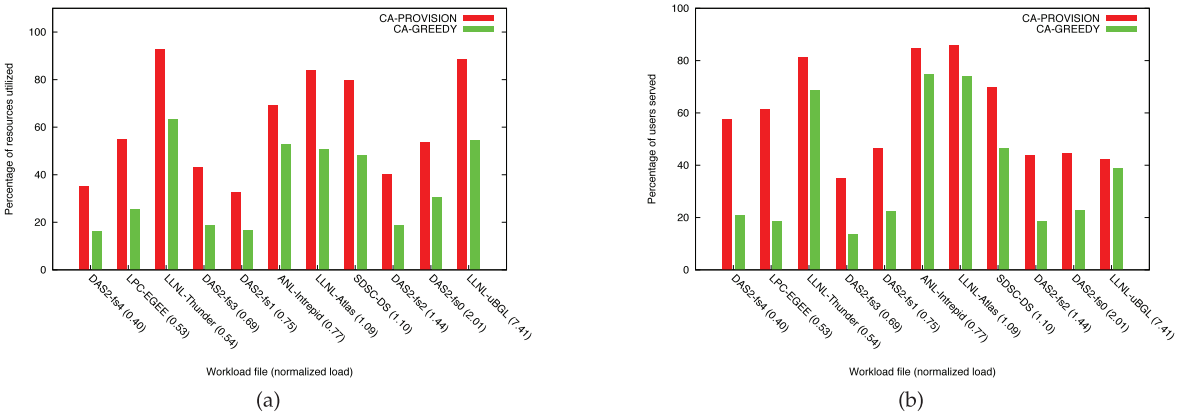


Fig. 2. CA-PROVISION versus CA-GREEDY: (a) Resource utilization; (b) percentage of users served. Horizontal axis shows the log file name with normalized load in parentheses.

We plot the average profit, the average revenue, and the average cost per processor-hour versus the workload logs in Figs. 1a, 1b, and 1c. In these figures, the workloads are sorted in ascending order of their normalized load. Note that the CA-PROVISION mechanism yields higher revenue in most of the cases. For workloads with normalized loads greater than 1.44, the revenue obtained by CA-PROVISION steadily increases exceeding that obtained by CA-GREEDY by up to 40 percent. This leads us to conclude that CA-PROVISION is capable of generating higher revenue, where there is high demand for resources.

In Fig. 1c, we observe that CA-PROVISION incurs a higher total cost for all workloads. Since CA-PROVISION decides about the number of VMs dynamically, it can allocate a higher number of VM instances than CA-GREEDY in an auction with identical bidders. This explains the higher cost incurred by CA-PROVISION; a unit VM instance costs c_I per unit time when idle, and $c_R > c_I$ per unit time while running (i.e., allocated to a user), as we assumed in Section 2. Therefore, by provisioning and allocating more VM instances, CA-PROVISION incurs higher costs to the cloud provider.

Now, the question is whether the interplay between increased revenue and increased cost can generate a higher profit. Utilizing more resources means serving more customers, hence selecting more bidders as winners. This interplay has two mutually opposite effects on the revenue. Obviously, increasing the number of winners has a positive effect on the revenue. On the other hand, selecting more winners pushes down their critical values, and thus, individual payments decrease. If the net effect is positive, we get a higher revenue and when it surpasses the increase in cost, we obtain a higher profit, and thus, achieve economies of scale. From Fig. 1a, we see that for normalized loads greater than 1.44, CA-PROVISION consistently generates higher profit than CA-GREEDY and the difference in profit grows rapidly. We also observe that for the workloads having load factors below 1.44, CA-PROVISION and CA-GREEDY obtain higher profit in equal number of cases. This suggests that for low loads the relative outcome of the mechanisms depends on other parameters.

In Figs. 2a and 2b, we compare the resource utilization and the percentage of served users obtained by the two mechanisms. CA-PROVISION achieves higher values for both utilization and percentage of served users. We want to draw the attention of the reader to the fact that in most of the cases the difference in utilization is around 30 percent. This is where we can improve a lot if we switch from static to dynamic provisioning and allocation. Since combinatorial auctions are already established tools for efficient allocation, combining them with dynamic provisioning can lead to a highly efficient resource allocation mechanism for clouds.

The number of users served is higher for CA-PROVISION because the VM instances are not statically provisioned. Therefore, a user requesting two VM_1 instances will not be left unallocated if there are no VM_1 instances available but a VM_2 instance is available as in the case of CA-GREEDY. Rather, CA-PROVISION "sees" the available resource as a computing resource equivalent to two VM_1 instances and will allocate this, for instance, to a user bidding for two VM_1

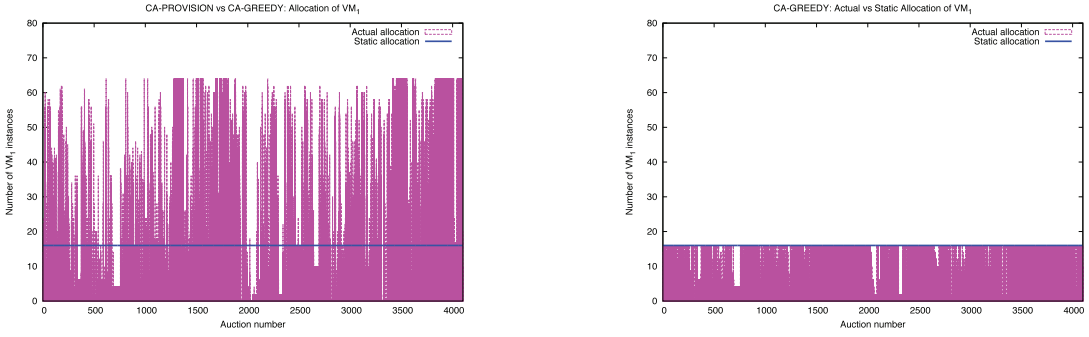


Fig. 3. Allocation of VM_1 instances: (a) by CA-PROVISION; (b) by CA-GREEDY. Workload file: DAS2-fs3-2003.

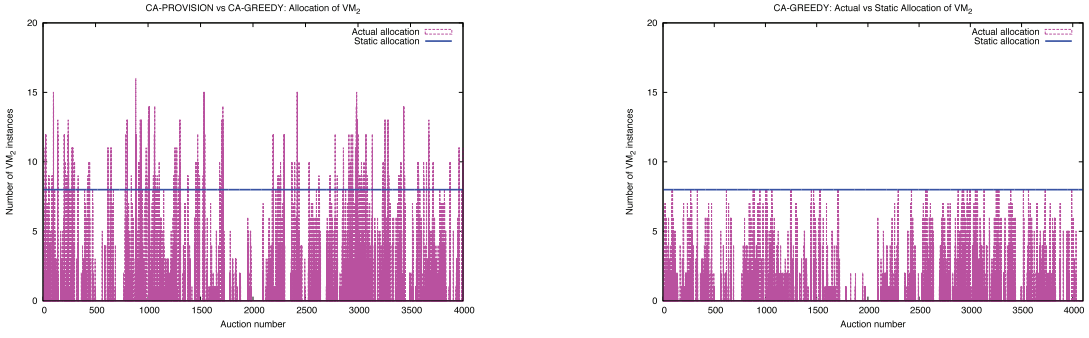


Fig. 4. Allocation of VM_2 instances: (a) by CA-PROVISION; (b) by CA-GREEDY. Workload file: DAS2-fs3-2003.

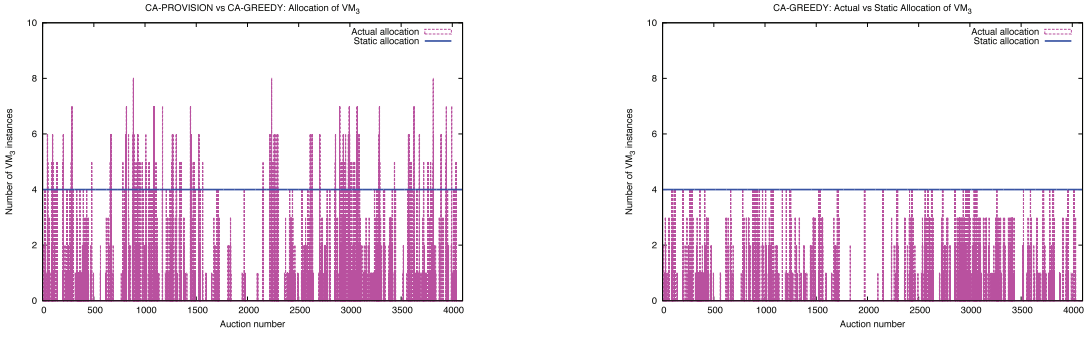
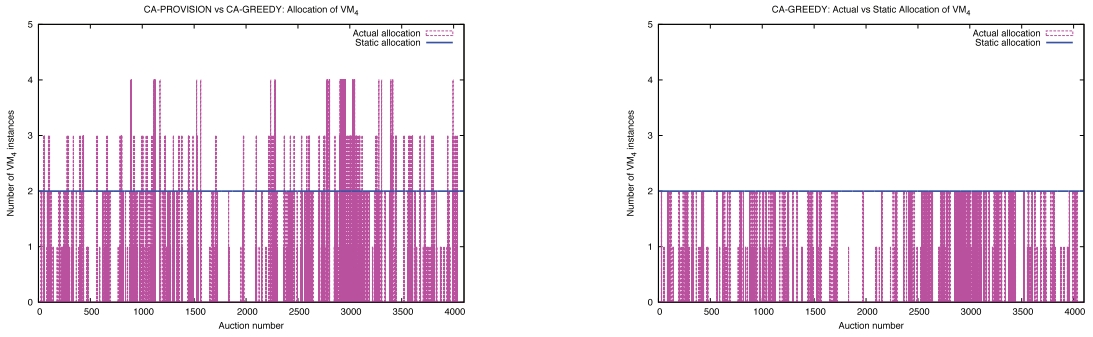


Fig. 5. Allocation of VM_3 instances: (a) by CA-PROVISION; (b) by CA-GREEDY. Workload file: DAS2-fs3-2003.

instances or a user bidding for one VM_2 instance, depending on whose reported valuation is higher. This approach increases the number of users served by CA-PROVISION.

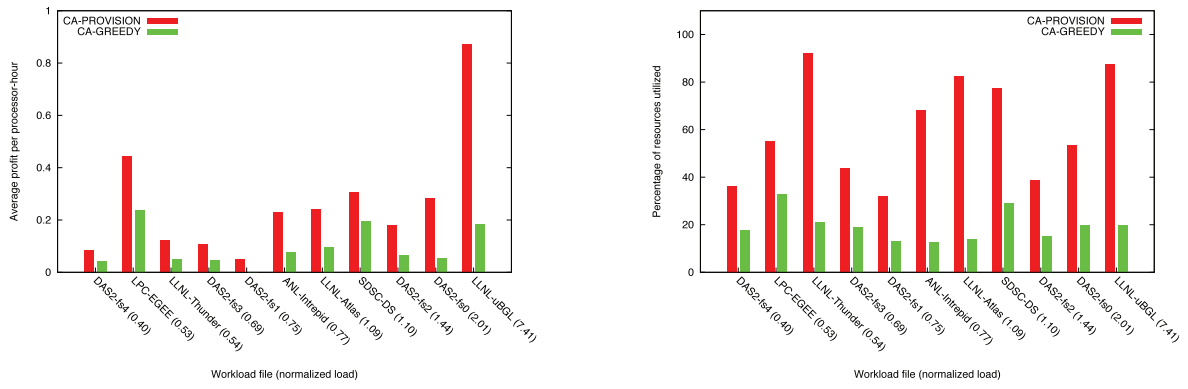
We now go into the details of the VM allocation by CA-GREEDY and CA-PROVISION for the DAS2-fs3-2003 workload. We pick a sample scenario from various combination of input parameters. In this experiment, the static VM allocation consists of 16 instances of type VM_1, eight instances of type VM_2, four instances of type VM_3, and two instances of type VM_4, This is equivalent to 64 instances of unit size (i.e., type VM_1). For this workload, a total of 4,100 auctions were held and in Figs. 3, 4, 5, and 6, we show the allocation of different VM instances in all these auctions. The figures corresponding to the CA-PROVISION mechanism show the number of the VM instances that are provisioned by the mechanism as thin vertical lines, where each line corresponds to an auction. For example, in Fig. 3a, we see that in many auctions, all 64 processors are configured as VM_1 instances. On the other hand, there are auction outcomes where no VM_1 instances are provisioned, as evident by the white strips touching the horizontal axis. The plots in the figures corresponding to the CA-GREEDY mechanism show (as thin vertical lines) the number of the VM instances that are allocated to the users. In both categories of plots, we show the static allocation line to compare the differences between static and dynamic provisioning.

Fig. 3a is particularly interesting because it shows that at times the demand for VM_1 goes far beyond what we would even think of allocating in advance. In some auctions, demands for VM_1 instances are much higher, and therefore, they push the allocation to the boundary. On the other hand, if we compare it with Fig. 3b, we see that CA-GREEDY indeed can capture the demand and allocate all 16 available instances of VM_1 in most of the auctions, but is limited to the availability of statically provisioned VMs.



(a) (b)

Fig. 6. Allocation of VM_4 instances: (a) by CA-PROVISION; (b) by CA-GREEDY. Workload file: DAS2-fs3-2003.



(a) (b)

Fig. 7. CA-PROVISION versus CA-GREEDY, where CA-GREEDY employs demand forecast over 10 percent of the past requests: (a) Average profit per processor-hour; (b) resource utilization. Horizontal axis shows the log file name with normalized load in parenthesis.

Eventually, it has to serve other less valued bids and loses revenue. Also, CA-GREEDY suffers from underallocation as it is clear from Figs. 5a and 5b. We see that the actual demand of VM_3 instances is lower than what we allocate statically (Fig. 5a) and the VM instances indeed remain unallocated in many cases (Fig. 5b).

In the last set of experiments, we evaluate CA-PROVISION against a modified version of CA-GREEDY that considers forecasting the demand of each type of VM instance (as described in Section 5.1.3). In Fig. 7a, we plot the overall profit for CA-GREEDY, where 10 percent of previous requests are used to generate the average number of VM instances to provision. We observe that even with demand forecasting CA-GREEDY cannot obtain the profit obtained by CA-PROVISION. This is due mainly to the large variations in the number of user requested VM instances in the log files considered for the experiments. Since similar performance is obtained for the cases in which the moving average is calculated over 25 and 50 percent of the previous user's requests, we do not present it here. When we examine the utilization obtained by the two mechanisms (Fig. 7b), we observe that CA-GREEDY with demand forecast is not able to obtain the level of utilization achieved by CA-PROVISION. Since the number of VM instances of each type varies greatly over time in the considered logs, the average over a window of time cannot capture well the demand and cannot improve the utilization of resources. The demand forecast using the moving average may improve the performance of CA-GREEDY in cases where the users' requests do not exhibit large variations over time.

We can summarize the experimental results as follows: The CA-GREEDY mechanism is capable of generating higher revenue than CA-PROVISION when there is matching demand with the supply. Also, in an auction where items are not "configurable," CA-GREEDY is a very efficient auction. But when we have reconfigurable items, as is the case in clouds, it is very hard to predict the demand very well in advance. In that case, CA-PROVISION is a better option and as today's technology supports, it can be deployed as a stand-alone configuration and allocation tool.

6 Conclusion

We addressed the problem of dynamically provisioning VM instances in clouds to generate higher profit, while determining the VM allocation with a combinatorial auction-based mechanism. We designed a mechanism called CA-PROVISION to solve this problem. We performed extensive simulation experiments with real workloads to evaluate our mechanism. The results showed that CA-PROVISION can effectively capture the market demand, provision the computing resources to match the demand, and generate higher revenue than CA-GREEDY, especially in high demand cases. In some of the low demand cases, CA-GREEDY performs better than CA-PROVISION in terms of profit but not in terms of utilization and percentage of

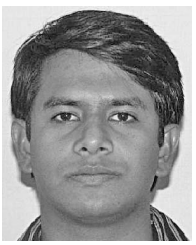
served users. We conclude that a highly effective VM instance provisioning and allocation system can be designed combining these two combinatorial auction-based mechanisms. We look forward to setting up a private cloud and implementing such a system in the near future.

ACKNOWLEDGMENTS

This paper is a revised and extended version of [31] presented at the third IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2011). This work was supported in part by the US National Science Foundation (NSF) grants DGE-0654014 and CNS-1116787.

REFERENCES

[1] Microsoft, "Windows Azure Platform," http://www.microsoft.com/windowsazure/, 2013.
[2] Amazon, "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/, 2013.
[3] R. Wang, "Auctions versus Posted-Price Selling," *The Am. Economic Rev.,* vol. 83, no. 4, pp. 838-851, 1993.
[4] N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani, *Algorithmic Game Theory.* Cambridge Univ. Press, 2007.
[5] S. Zaman and D. Grosu, "Combinatorial Auction-Based Allocation of Virtual Machine Instances in Clouds," *Proc. IEEE Second Int'l Conf. Cloud Comp. Technology and Science,* pp. 127-134, 2010.
[6] D.G. Feitelson, "Parallel Workloads Archive: Logs," http://www.cs.huji.ac.il/labs/parallel/workload/logs.html, 2013.
[7] Amazon, "Amazon EC2 Instance Types," http://aws.amazon.com/ec2/instance-types/, 2013.
[8] P. Shivam, A. Demberel, P. Gunda, D. Irwin, L. Grit, A. Yumerefendi, S. Babu, and J. Chase, "Automated and On-Demand Provisioning of Virtual Machines for Database Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1079-1081, 2007.
[9] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma, "Towards Autonomic Workload Provisioning for Enterprise Grids and Clouds," *Proc. IEEE/ACM 10th Int'l Conf. Grid Computing,* pp. 50-57, 2009.
[10] C. Vecchiola, R.N. Calheiros, D. Karunamoorthy, and R. Buyya, "Deadline-Driven Provisioning of Resources for Scientific Applications in Hybrid Clouds with Aneka," *Future Generation Computer Systems,* vol. 28, pp. 58-65, 2012.
[11] R. Wolski, J.S. Plank, J. Brevik, and T. Bryan, "Analyzing Market-Based Resource Allocation Strategies for the Computational Grid," *The Int'l J. High Performance Computing Applications,* vol. 15, no. 3, pp. 258-281, 2001.
[12] A. Das and D. Grosu, "Combinatorial Auction-Based Protocols for Resource Allocation in Grids," *Proc. 19th Int'l Parallel and Distributed Processing Symp., Sixth Workshop Parallel and Distributed Scientific and Eng. Computing,* 2005.
[13] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, "Distributed Systems Meet Economics: Pricing in the Cloud," *Proc. Second USENIX Workshop Hot Topics in Cloud Computing,* 2010.
[14] J. Altmann, C. Courcoubetis, G.D. Stamoulis, M. Dramitinos, T. Rayna, M. Risch, and C. Bannink, "GridEcon: A Market Place for Computing Resources," *Proc. Workshop Grid Economics and Business Models,* pp. 185-196, 2008.
[15] M. Risch, J. Altmann, L. Guo, A. Fleming, and C. Courcoubetis, "The GridEcon Platform: A Business Scenario Testbed for Commercial Cloud Services," *Proc. Workshop Grid Economics and Business Models,* pp. 46-59, 2009.
[16] Amazon, "Amazon EC2 Spot Instances," http://aws.amazon.com/ec2/spot-instances/, 2013.
[17] O.A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon Ec2 Spot Instance Pricing," *Proc. IEEE Third Int'l Conf. Cloud Computing Technology and Science,* 2011.
[18] S. Wee, "Debunking Real-Time Pricing in Cloud Computing," *Proc. IEEE/ACM 11th Int'l Symp. Cluster, Cloud and Grid Computing,* pp. 585-590, 2011,
[19] Q. Zhang, Q. Zhu, and R. Boutaba, "Dynamic Resource Allocation for Spot Markets in Cloud Computing Environments," *Proc. IEEE Fourth Int'l Conf. Utility and Cloud Computing,* pp. 178-185, 2011.
[20] M.H. Rothkopf, A. Pekec, and R.M. Harstad, "Computationally Manageable Combinatorial Auctions," *Management Science,* vol. 44, no. 8, pp. 1131-1147, 1998.
[21] T. Sandholm, "Algorithm for Optimal Winner Determination in Combinatorial Auctions," *Artificial Intelligence,* vol. 135, nos. 1/2, pp. 1-54, 2002.
[22] E. Zurel and N. Nisan, "An Efficient Approximate Allocation Algorithm for Combinatorial Auctions," *Proc. Third ACM Conf. Electronic Commerce,* pp. 125-136, 2001.
[23] P. Cramton, Y. Shoham, and R. Steinberg, *Combinatorial Auctions.* MIT Press, 2005.
[24] D. Lehmann, L.I. O'Callaghan, and Y. Shoham, "Truth Revelation in Approximately Efficient Combinatorial Auctions," *J. the ACM,* vol. 49, no. 5, pp. 577-602, 2002.
[25] A. Mu'alem and N. Nisan, "Truthful Approximation Mechanisms for Restricted Combinatorial Auctions," *Proc. 18th Nat'l Conf. Artificial Intelligence,* pp. 379-384, 2002.
[26] P. Briest, P. Krysta, and B. Vöcking, "Approximation Techniques for Utilitarian Mechanism Design," *SIAM J. Computing,* vol. 40, no. 6, pp. 1587-1622, 2011.
[27] C. Chekuri and I. Gamzu, "Truthful Mechanisms via Greedy Iterative Packing," *Proc. 12th Workshop Approximation Algorithms for Combinatorial Optimization Problems,* pp. 56-69, 2009.
[28] U. Lampe, M. Siebenhaar, A. Papageorgiou, D. Schuller, and R. Steinmetz, "Maximizing Cloud Provider Profit from Equilibrium Price Auctions," *Proc. IEEE Fifth Int'l Conf. Cloud Computing,* pp. 83-90, 2012.
[29] V. Prasad, S. Rao, and A. Prasad, "A Combinatorial Auction Mechanism for Multiple Resource Procurement in Cloud Computing," *Proc. 12th Int'l Conf. Intelligent Systems Design and Applications,* pp. 337-344, 2012.
[30] D.G. Feitelson, "Parallel Workloads Archive: Standard Workload Format," http://www.cs.huji.ac.il/labs/parallel/workload/swf.html, 2013.
[31] S. Zaman and D. Grosu, "Combinatorial Auction-Based Dynamic VM Provisioning and Allocation in Clouds," *Proc. IEEE Third Int'l Conf. Cloud Computing Technology and Science,* pp. 107-114, 2011.



Sharrukh Zaman received the bachelor's of computer science and engineering degree from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh. He is currently working toward the PhD degree in the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests include cloud computing, distributed systems, game theory, and mechanism design. He is a student member of the IEEE and the IEEE Computer Society.

Daniel Grosu received the diploma in engineering (automatic control and industrial informatics) from the Technical University of Iaşi, Romania, in 1994 and the MSc and PhD degrees in computer science from the University of Texas at San Antonio in 2002 and 2003, respectively. Currently, he is an associate professor in the Department of Computer Science, Wayne State University, Detroit. His research interests include distributed systems and algorithms, resource allocation, computer security and topics at the border of computer science, game theory, and economics. He has published more than 80 peer-reviewed papers in the above areas. He has served on the program and steering committees of several International meetings in parallel and distributed computing. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society.