Computer Networks 94 (2016) 231–249

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Auction-based cloud service differentiation with service level objectives

Jianbing Ding [a,b,*], Zhenjie Zhang [c], Richard T. B. Ma [d], Yin Yang [e]

[a] School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
[b] SYSU-CMU Shunde International Joint Research Institute, Shunde, China
[c] Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore
[d] School of Computing, National University of Singapore, Singapore
[e] College of Science and Engineering, Hamad Bin Khalifa University, Qatar

ARTICLE INFO

Article history:
Received 28 April 2015
Revised 14 August 2015
Accepted 4 November 2015
Available online 10 November 2015

Keywords:
Cloud computing
Service differentiation
Auction
MapReduce

ABSTRACT

The emergence of the cloud computing paradigm has greatly enabled innovative service models, such as Platform as a Service (PaaS), and distributed computing frameworks, such as MapReduce. However, most existing cloud systems fail to distinguish users with different preferences, or jobs of different natures. Consequently, they are unable to provide *service differentiation*, leading to inefficient allocations of cloud resources. Moreover, contentions on the resources exacerbate this inefficiency, when prioritizing crucial jobs is necessary, but impossible. Motivated by this, we propose *Abacus*, a generic resource management framework addressing this problem. Abacus interacts with users through an *auction mechanism*, which allows users to specify their priorities using *budgets*, and job characteristics via *utility functions*. Based on this information, Abacus computes the optimal allocation and scheduling of resources. Meanwhile, the auction mechanism in Abacus possesses important properties including incentive compatibility (i.e., the users' best strategy is to simply bid their true budgets and job utilities) and monotonicity (i.e., users are motivated to increase their budgets in order to receive better services). In addition, when the user is unclear about her utility function, Abacus automatically learns this function based on statistics of her previous jobs. Extensive experiments, running Hadoop on a private cluster and Amazon EC2, demonstrate the high performance and other desirable properties of Abacus.

1. Introduction

Cloud computing is a cost-effective paradigm for both the cloud providers and the users. The providers benefit by effectively multiplexing dynamic user demands of various computing resources, e.g., CPU, storage, bandwidth, etc., through virtualization techniques. Meanwhile, the users are liberated from large capital outlays in hardware deployment and maintenance. Successful cloud models include Infrastructure as a Service (e.g., Amazon's EC2), and data-intensive distributed computing paradigm (e.g., Map-Reduce), both of which are well adopted for a wide spectrum of web services and data management tasks [13,15].

Cloud systems can be categorized into private clouds, which are used exclusively by one organization, and public ones, which rent out their computational capacities to customers. For private clouds, one of the most important objectives is *efficiency*, meaning that the overall utility derived from the jobs executed with the limited cloud resources. Although public clouds might put profitability before efficiency, both objectives are often aligned and achieved by serving the most valued jobs under resource competition. In order to maximize efficiency, jobs should be

* Corresponding author. Tel.: +86 18925190471.
E-mail addresses: dingsword@gmail.com (J. Ding), zhenjie@adsc.com.sg (Z. Zhang), tbma@comp.nus.edu.sg (R. T. B. Ma), yyang@qf.org.qa (Y. Yang).

http://dx.doi.org/10.1016/j.comnet.2015.11.007
1389-1286/

differentiated based on their characteristics, including the utilities they generate and the distinct resources they require. For instance, computation-intensive jobs may need powerful CPUs more than other resources, whereas bandwidth is often the most important resource for delay-sensitive applications. Intuitively, resources should be allocated to jobs of high importance, and to jobs that need them the most. Existing cloud systems generally do not provide adequate capability for such service differentiation. In particular, private clouds mainly use simple resource allocation strategies, such as first-in-first-out and fair-share. Public clouds essentially differentiate users based on the amount of money they pay for each type of resources. For instance, Amazon EC2 bundles resources into virtual machines (VMs), and each type of VM has its unique configuration and unit-time price. Based on these prices and the status of their applications, the users decide by themselves the type and number of VM-hours to purchase. Moreover, such prices for computational resources in public clouds often fluctuate continuously, forcing users to monitor these prices and adjust their VM portfolios accordingly, if they want to maximize overall utility within budget limits.

While the above pricing scheme can be seen as a kind of manual service differentiation, to our knowledge, currently there is no solution for *automatic* service differentiation. Providing this functionality is challenging in several aspects. First, only users (possibly) know the utilities and resource usage patterns of their own jobs; hence, the cloud system needs an effective way to obtain this information from the users. Second, an *incentive compatible* mechanism is needed to prevent any user from misreporting information so as to increase its own utility, as this may hurt the performance of other jobs as well the overall utility of the entire cloud. Third, realizing an abstract service differentiation solution on a real cloud system is non-trivial, as the implementation must work seamlessly with the existing resource allocation and scheduling modules.

Facing these challenges, we propose a novel cloud resource allocation framework called *Abacus*, which enables service differentiation for clouds. Abacus acquires information on users' job characteristics through a novel auction mechanism. Resources are then dynamically allocated according to the auction results and the availability of each type of resources. Abacus' auction mechanism is *incentive-compatible*, meaning that every user's dominating strategy is to simply bid its true job characteristics. This also implies that the auction is *stable*, i.e., no user can benefit from unilateral bid changes. These properties ensure that as long as a user's job characteristics remain the same, there is no need to monitor the auction or to change bids. In this aspect, Abacus is much easier to use compared to the price-based service differentiation as in Amazon EC2. Further, the auction is *monotonic*, which guarantees fairness in the sense that users paying more for a type of resource are always allocated higher quantities of it. Finally, Abacus is *efficient*, which achieves high overall system utility under the above constraints, as confirmed by our experimental evaluations.

Abacus can be used in various cloud systems, including public and privacy ones, and clouds running on different platforms. To demonstrate the practical impact of Abacus, we implement it on top of Hadoop, and evaluate its effectiveness and efficiency using Map-Reduce workloads. To further improve the usability of Abacus, especially for cloud system users without clear knowledge on the utility model of their own *repeated* jobs, we extend our standard auction mechanism to enable users to submit bids with only budget information. After running the jobs for a number of rounds under default utility functions, the utility prediction component is capable of recovering the true utility function, using regression techniques. Experiments using a large-scale cluster confirm that Abacus successfully achieves high overall utility, high performance, and all the designed desirable properties.

The main contributions of the paper are listed below

- We present a new study on service differentiation techniques for general cloud system. Our solution potentially opens new business models for cloud systems in the future, and enables ordinary users to exploit the benefits of clouds.
- We propose *Abacus*, an auction based approach to cloud system resource allocation and scheduling, with enticing features such as *incentive-compatibility*, *system stability* and *system efficiency*.
- We simplify the auction procedure by allowing the users to skip the utility function when the user is unsure or unaware of the exact utility model of his own repeated jobs.
- We implement Abacus by modifying the scheduling algorithm in Hadoop, and test it on a large-scale cloud platform. Our experimental results verify the truthfulness of our auction-based mechanism, system efficiency, as well as the accuracy of our utility prediction algorithm.

A preliminary version of Abacus appears in [41]. The main difference between [41] and the full version is that the former mainly focuses on the Abacus model and its *theoretical properties*, whereas the latter also presents solutions and results that are crucial for applying Abacus *in practice*. These include (i) a novel algorithm to handle jobs with Service Level Objectives (SLOs), presented in Section 6, which enables Abacus to support jobs running on cloud systems with performance requirements, e.g. maximum response time, (ii) a large-scale experimental evaluation of Abacus on a public cloud (i.e., Amazon EC2) with real-world workloads, presented in Section 7.2, (iii) performance comparison between Abacus and ARIA [37], a state-of-the-art solution for SLO-based scheduling, also presented in Section 7.2. In addition, the full version provides detailed proofs for our theoretical results on the robustness and soundness of the proposed algorithms.

In the following, Section 2 reviews related work. Section 3 provides problem definition and assumptions. Section 4 details the auction mechanism. Section 5 discusses automatic optimization for users not knowing their own utility functions. Section 6 extends Abacus to handle jobs with Service Level Objective. Section 7 contains an extensive experimental evaluation. Finally, Section 8 concludes the paper.

2. Related work

2.1. Grid/Cloud resource allocation

Resource allocation [12,25] and scheduling [17,27] in distributed systems have been extensively studied in the

networking community. Most previous work focuses on allocating bandwidth among competing network flows. For instance, fair scheduling algorithms, e.g., Weighted Fair Queueing (WFQ) [17] and Generalized Process Sharing (GPS) [27], achieve a *max–min fair* [12] allocation. Service differentiations can be provided via these scheduling schemes, when the priorities of the service classes that the jobs belong to are known to the schedulers. In the context of cloud systems, however, the priority of the jobs are usually not known in advance. Consequently, the above solutions no longer apply. In fact, although cloud computing has received increased attention within both the IT industry [1,3,5] and the research community [10], little work addresses the issue of providing service differentiations to users.

Different allocation algorithms are proposed to better meet the requirements of multiple application running over virtualized/grid environment. Stillwell et al. [31] propose to use *yield* and *stretch* as metric on resource allocation and performance, along with optimization strategies to maximize the performance of all applications. Our utility definition is a further generalization of their concepts, and is thus applicable on wider domains. Raicu et al. [29] present a data-aware scheduling method, focusing data-intensive applications with the processing logic separated from the data storage layer. Yang et al. [39] discuss a robust scheduler considering the variance of the computation capacity of the distributed nodes instead of the expected performance. Note that all these works assume static workload during the run-time of the resource utilization.

Hindman et al. [23] propose *Mesos*, a resource allocation system that manipulates resources in a cloud system among different computation platforms, e.g. Hadoop, HBase and MPI. In Mesos, every platform submits its resource requests to the master node, which then makes offers of the resources to the platforms based on the remaining resources available. Each platform can either accept an offer and the corresponding resources, or reject it in anticipation for a better offer in the future. Unfortunately, Mesos does not support any type of service differentiation. Popa et al. [28] extend Mesos to address bandwidth assignment in cloud systems. Since bandwidth is the major bottleneck for communication-intensive applications, they apply new strategies for bandwidth assignment that ensures both fairness and efficiency. This work does not discuss service differentiation, either. There exist other practical approaches to the resource allocation problem on cloud, e.g. [9,32,43], but without theoretical analysis on the effectiveness in competitive environment.

Regarding application profiling, Urgaonkar et al. [35] utilizes OS kernel-based profiling to model the target applications when running on the VMs. They also show that under-provision the applications brings benefit on overall utilization of the computation resource. Meyer et al. [26] show the possibility of understanding the workflow jobs when they are repeated in the distribution system. We apply similar strategy in utility function estimation, which is much more complicated on computing the expected utility under any resource provisioning.

2.2. Map-Rreduce scheduling

MapReduce [16] is among the most popular paradigms for data-intensive computation in cloud systems, and its open source implementation Hadoop [2] is commonly used for various applications. Scheduling is a fundamental problem in MapReduce. So far, existing work on MapReduce scheduling focuses mainly on system performance; to our knowledge, no existing scheduling solutions can achieve service differentiation for MapReduce with multiple resource types.

Zaharia et al. [40] propose a scheduler that maintains a queue for jobs requesting particular Map or Reduce nodes. They propose two techniques delay, scheduling and copy-compute, which provide efficiency and fairness in a multi-user environment. Our work is orthogonal to theirs, and can be seen as a meta-scheduler for the MapReduce jobs that achieves service differentiation. In fact, our current implementation of Abacus is based on the *Fair Scheduler* proposed in [40]. Issard et al. [24] independently propose a scheduling strategy similar to [40]. Their method utilizes the queueing strategy for resource allocation, which is optimized by a max-flow model on tasks from all jobs. Moreover, their scheduler allows users to specify different policies, such as fair sharing with preemption.

Sandholm and Lai [30] attempt to provide service differentiation for MapReduce, based on user priorities and a proportional resource allocation mechanism. Abacus differs from their work in two important aspects. First, Abacus manages multiple types of resources, whereas [30] can only handle one type of resource. Second, unlike [30], Abacus is not limited to MapReduce or any particular cloud computing platform. Finally, Abacus can be parameterized to balance fairness and efficiency of the system. Thus, Abacus can be viewed as a generalization of [30].

Herodotou et al. [21,22] explore the problem of performance estimation, without direct optimization on scheduling algorithm. They derive general models to predict the performance of MapReduce jobs when certain resources are assigned to the job. This model enables the users to choose appropriate amount of resource to purchase before starting the job. *Abacus* can also solve the problem with a much simpler: the users only need to submit their budgets and the system automatically optimizes the resource allocation for all users.

2.3. Auction mechanism

Recently, auctions have been successfully applied to online keyword advertising. Specifically, each advertiser submits a bidding price for each of her interested keywords. After a user issues a query, the search engine returns the search results, along with a number of ads, which are selected based on the auction results. Existing studies have shown that the revenue of such keyword auctions heavily depends on the auction mechanism [18,36]. This has propelled the research from both economics and computer science communities [7,8,20,26], attempting to identify effective and efficient combinations of computational and economical models. Currently, most keyword advertising systems adopt second-price auction mechanisms, e.g.,

Vickrey–Clarke–Groves (VCG) [38] and Generalized Second Price (GSP) [36]. Several studies also consider budget constraints for the advertisers, e.g., [11,14,19]. Unfortunately, auction mechanisms designed for online advertising are not directly applicable to cloud resource allocation, for several reasons. First, a user needs cloud resources only when it has jobs to run. Hence, traditional fixed assignment strategy may lead to waste of resources, when a user holding resources does not have jobs to use them. Second, the utility functions in our setting are more complicated than online advertising market. In particular, a job usually requires multiple types of resources. Next we describe the proposed solution Abacus, which solves these problems.

3. Overview of ABACUS

Abacus (Auction-BAsed CloUd System) is a general framework for managing multiple types of system resources. Assume that there are m different types of resources. For each resource type j, there are a finite number of identical units of the resource. In existing cloud systems, the cloud users must buy the resources before starting their jobs. In Abacus, the resource allocation is done on the fly, based on the profiles of the jobs currently running in the system. It is conceptually consistent with the new generation of cloud scheduler, which is expected to allocate resources for heterogenous applications under a unified framework.

Abacus allows each user to submit multiple jobs at any time; when a job finishes, it is automatically removed from Abacus. Similar to existing resource managers, e.g., MapReduce fair scheduler [4] and Spark scheduler [6], whenever there is a change in the set of jobs managed by Abacus (i.e., when a job arrives or finishes), Abacus re-runs the resource allocation module. As we explain soon, resource allocation in Abacus mainly involves an auction algorithm (described in Section 4), which is highly efficient as shown in our experiments in Section 7.1.1. Thus, Abacus can effectively support dynamic job arrivals and completions.

Each job submission J_i consists of two parts, $J_i = (b_i, u_i)$, in which b_i is the budget the user is willing to pay for the job and u_i is a utility function indicating the benefit of the job when the job is allocated with a certain amount of resources. We denote $s_i = (s_{i1}, s_{i2}, \ldots, s_{im})$ as the vector of resource allocation probabilities assigned to job J_i, in which each s_{ij} is a probability associated with resource j in [0, 1]. The utility of the job J_i is evaluated by the submitted utility function $u_i(s_i)$. Formally, the utility function u_i is a mapping $u_i : [0, 1]^m \mapsto \mathbb{R}$. In this paper, we focus on utility functions in the form of *sum of non-decreasing concave functions*, i.e. $u_i(s_i) = \sum_{j=1}^{m} g_j(s_{ij})$, where each $g_j(s_{ij})$ is non-decreasing and concave.

To better illustrate the concept of utility functions, we present two concrete examples. One is sum of *linear utility functions*. Given a non-negative weight w_j for each resource type j, a utility function takes the form $g_j(s_{ij}) = w_j s_{ij}$. The overall utility of the job is then the weighted sum on the resources assigned to job J_i, i.e., $u_i(s_i) = \sum_{j=1}^{m} w_j s_{ij}$. Linear utility function is suitable for computation models with *substitutable* resource.

Example 1. Web service. in Amazon EC2, the user may want to find virtual machines with large amounts of main memory to run his web service. The service is still usable, if EC2 assigns the user with machines with small memory, rendering a lower level of service quality. Therefore, the utility function on EC2 can be written as a linear utility function, with s_{ij} as the probability of getting VM of type j and w_j being performance of VM of type j running the web service. The job gains positive utility, even when there is no resource assigned on certain type, i.e. $u_i(s_i) > 0$ even if $s_{ij} = 0$ for some j.

Another important class of functions that satisfy our requirements is *logscale utility functions* of them form $g_j(s_{ij}) = w_j \log s_{ij}$. The overall utility of a job is then $u_i(s_i) = \sum_{j=1}^{m} w_j \log s_{ij}$. The logscale utility function differs from the linear utility function in that the former returns negative infinity if any $s_{ij} = 0$. Thus, the logscale utility function is a better model for jobs that require all types of computation resources.

Example 2. MapReduce: in MapReduce, each job is processed by two types of operators, i.e. *map* and *reduce*, with specified number of required resources on each of the operators. The job cannot be finished if the system does not allow the job to use either type of the operators. The utility of a MapReduce job is the reverse of the running time, i.e. the shorter completion time the better utility. In this utility model, s_{i1} and s_{i2} are the probability of getting map and reduce operators during job running, and w_1 and w_2 indicate computation workloads of map and reduce tasks. It is thus more appropriate to model MapReduce jobs using log-scale utility function, with the Map and Reduce operators as two types of resources.

Note that the results in this paper are valid for both linear and logscale utility functions, and, in general, any utility function satisfying the condition of *sum of non-decreasing concave functions*.

In both of the examples above, cloud service providers currently only support direct purchase of computation resource, e.g. machine-hours, to run their applications. By employing Abacus, the user only submits his job J_i to Abacus, and Abacus completely handles the resource allocation on behalf of the active users in the system. The major design goal of the auction-based scheduling strategy used in Abacus is to prioritize the jobs, when a large number of jobs are competing for the limited computation resource.

In particular, all of the current jobs are kept in the runtime scheduler in Abacus. Assume that there are n concurrent jobs running in the system with profiles $\{J_1, J_2, \ldots, J_i, \ldots, J_n\}$, Abacus calculates the resource assignment vector s_i for each J_i. There are m resource request queues maintained in Abacus. Each queue Q_j stores the running jobs currently waiting for the resource of type j. Given the assignment vectors, $\{s_1, \ldots, s_i, \ldots, s_n\}$, when a particular resource of type j is available for a new task, Abacus assigns the resource to job J_i with probability s_{ij}. It is important to emphasize that tasks could run independently with a single unit of resource. Therefore, there is no deadlock in the system, in which some jobs with low priority are starving when waiting for other jobs.

There are two important components in Abacus, *Auctioneer* and *Scheduler*. Fig. 1 presents the relationship between these components, in the context of the MapReduce

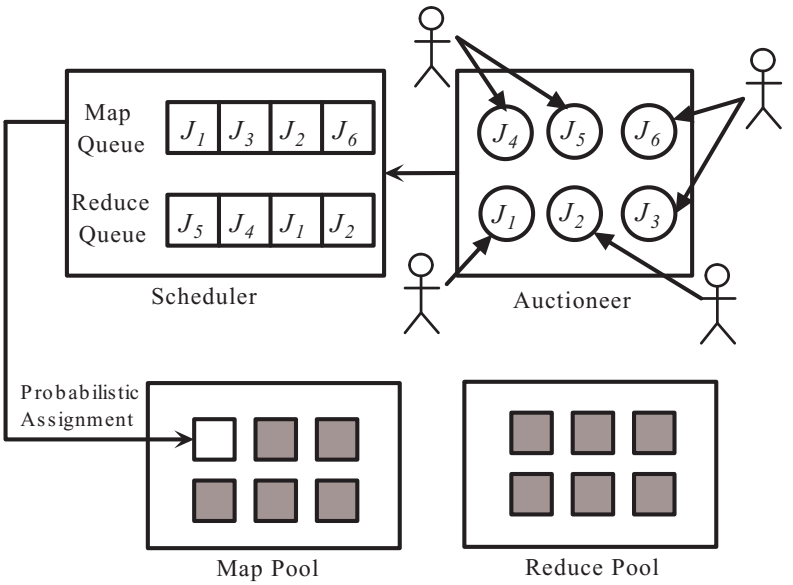


Fig. 1. Abacus system architecture on Hadoop.

framework. The auctioneer is responsible for the scheduling probability assignment. When jobs are added or removed from the system, the auctioneer recalculates the probability assignment vectors for the scheduler on all types of resources. To support jobs in MapReduce, there are two queues for map nodes and reduce nodes respectively. Given the probabilities derived by the auctioneer, the scheduler selects the next job waiting for certain resource according to the probabilities. This selection procedure runs again when one of the task finishes the computation and returns the resource to the scheduler.

In our example, there is an idle map node. The scheduling component thus picks up a job among $\{J_1, J_2, J_3, J_6\}$ to run a new map task on the idle node. When the cloud system finishes a job J_i, it charges the user according to his bidding price/budget b_i. Abacus is expected to provide better services, i.e., more computation resources for a job, if its bidding budget is higher. This mainly relies on the auction mechanism employed by the auctioneer. Unlike existing solutions on dynamic resource assignment, e.g., Amazon EC2 and Mesos [23], Abacus emphasizes the economic effects on the resource allocation. Instead of considering only the requirements from the jobs, the system balances between the system efficiency and fairness. Generally speaking, users with higher budget is given higher priority in job running, while users with low budget remains active in the system without starving.

4. Auction-based protocol

4.1. Basic protocol

Recall from Section 3 that every user submits his job along with profile J_i. Assume that $\mathbb{B}$ is the domain consisting of all valid job submissions. Our auction algorithm calculates an assignment matrix, indicating the priority of job J_i with respect to resource of type j, for every pair of i and j, i.e. $S = (s_{ij})$ of size $n \times m$. In this matrix, every s_{ij} is a non-negative real number such that $\sum_i s_{ij} = 1$ for every resource type j. We use $\mathbb{S}$ to denote the domain of all matrices meeting these constraints. Based on these definitions, the auction mechanism is a function M mapping from the domain of job profiles to the assignment matrix domain, i.e. $M : \mathbb{B}^n \mapsto \mathbb{S}$.

In Fig. 2, we present an example auction with three jobs running on a Hadoop system. The profile of job J_1 is ($100, $u_1(x, y) = 3x + 2y$). where x and y denote the number of map and reduce nodes, respectively. Similarly, jobs J_2 and J_3 are associated with different budgets and other linear utility functions. Based on the specific auction mechanism, the auction component outputs an assignment matrix on the right side of the figure. The job J_1, for example, has probabilities $s_{11} = 0.310$ and $s_{12} = 0.236$ to get map nodes and reduce nodes, respectively. An interesting observation is that J_3 gains more resources on reduce nodes than J_2, even though its overall budget is smaller. This is because the system understands that J_3's job utility depends on reduce resource more than J_2 does, which is implicitly expressed in J_3's utility function $u_3(x, y) = 2x + 4y$.

4.2. Auction mechanism

To compute the assignment matrix, the auction component virtually partitions the budget b_i of job J_i into small sub-budgets, i.e. $\{b_{i1}, \ldots, b_{im}\}$, such that $\sum_j b_{ij} = b_i$. Each sub-budget b_{ij} is part of b_i for job J_i to spend on resource j. Given the virtual partitions on all jobs in the system, the probability of job J_i on resource j is calculated based on the following equation.

$$s_{ij} = \frac{(b_{ij})^\alpha}{\sum_{l=1}^{n} (b_{lj})^\alpha} = \frac{(b_{ij})^\alpha}{(b_{ij})^\alpha + \sum_{l \neq i} (b_{lj})^\alpha} \quad (1)$$

Here, α is a non-negative scaling factor, balancing the priorities of high-budget jobs and fairness between jobs. When $\alpha = 0$, the assignment probability is uniform on all users, i.e.

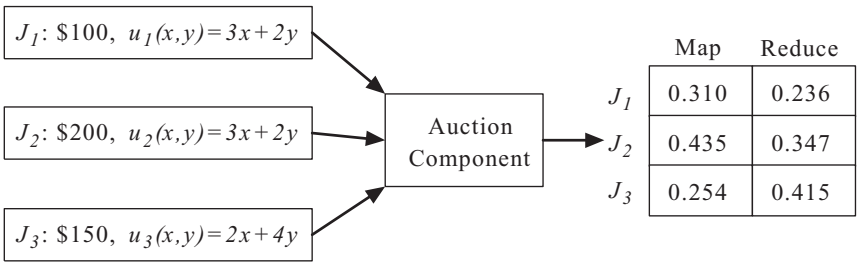


Fig. 2. Example of auction with three jobs on Hadoop.

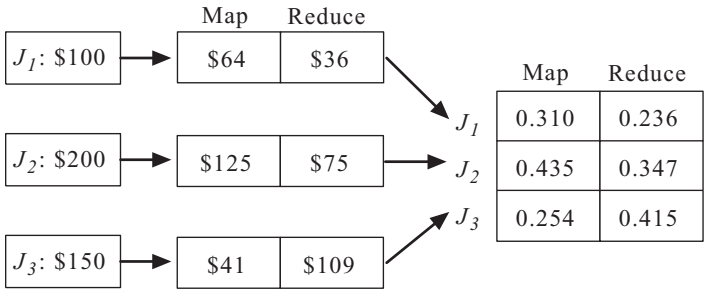


Fig. 3. Example of sub-budget partitioning.

$s_{ij} = 1/n$ for any i and j, regardless of the budgets of the jobs. When α approaches infinity, the job with the highest budget dominates the probability and all other jobs are given almost zero resource of type j. In Abacus, however, we only allow α to be a real number between 0 and 1, to ensure the following important property.

Lemma 1. *When $\alpha \in [0, 1]$, the partial derivative $\frac{\partial u_i}{\partial b_{ij}}$ is a monotonically decreasing function with respect to b_{ij}.*

Intuitively, the lemma above implies that the marginal utility of a job on a certain resource shrinks when more sub-budget is devoted to the resource. This property turns out to be crucial in our following analysis.

In Fig. 3, Abacus partitions the budgets based on the jobs in Fig. 2. Abacus helps job J_1, for example, to assign \$64 and \$36 on map and reduce resources respectively. By setting $\alpha = 0.5$, we can easily verify the correctness of the final assignment matrix on the right side, with the sub-budgets on all jobs.

Based on the assignment rule, a job J_i prefers to put more sub-budgets on resources that contribute more to its utility function. However, Abacus does not allow the user to manually adjust virtual sub-budgets themselves. Instead, the auctioneer automatically optimizes the sub-budgets for all jobs. Although it is possible to support sub-budget specification by user, it may lead to violations to the desirable properties, e.g. incurring cycling chaos [18]. The details of the analysis will be covered in the rest of this section.

The auction mechanism employed in Abacus aims to find a probability matrix $S = M(\{J_1, \ldots, J_n\})$ based on our resource allocation mechanism M. In Algorithm 1, we list the pseudocode for the assignment mechanism M. Basically, the algorithm initializes the sub-budgets by evenly partitioning the budget on all resources, and then applies the round-robin optimization for every job J_i. Given the current sub-budgets from all other jobs, the auction mechanism finds a new sub-budget combination $\{b_{i1}^{(t)}, \ldots, b_{im}^{(t)}\}$ on all resource types in the current round. The algorithm terminates until all jobs agree on their sub-budgets, i.e. there is no change on the sub-budgets in two consecutive iterations.

Algorithm 1 Auction $(\{J_1, J_2, \ldots, J_n\})$.

1: Calculate the initial sub-budget matrix $b^{(1)}$ that $b_{ij}^{(1)} = \frac{b_i}{m}$ for each resource j.
2: Let $t = 1$
3: **while** $\exists i, j,\ \|b_{ij}^{(t)} - b_{ij}^{(t-1)}\|_2 \geq \epsilon$ **do**
4: **for** each b_i **do**
5: Run Algorithm 2 to find out the best sub-budget combination $\left(b_{i1}^{(t+1)}, \ldots, b_{im}^{(t+1)}\right)$ for job J_i
6: Assemble new assignment matrix $\left\{b_{ij}^{(t+1)}\right\}$ by combining sub-budgets from all jobs
7: Let t increment by 1
8: Return final assignment matrix $\{s_{ij}\}$ based on $\left\{b_{ij}^{(t)}\right\}$

To calculate the best sub-budgets for a single job J_i, our Algorithm 2 utilizes the concave property of the utility

Algorithm 2 OptimizeJob $(i, \{b_{i1}, \ldots, b_{im}\})$.

1: Calculate $s_{ij} = \frac{(b_{ij})^\alpha}{\sum_{l=1}^{n}(b_{lj})^\alpha}$ for every j
2: **for** each resource j **do**
3: Calculate $b'_{ij} = \frac{\frac{\partial u_i}{\partial s_{ij}} s_{ij}(1-s_{ij})}{\sum_{k=1}^{n}\left(\frac{\partial u_k}{\partial s_{kj}} s_{kj}(1-s_{kj})\right)} b_i$
4: Return $\{b'_{i1}, \ldots, b'_{im}\}$

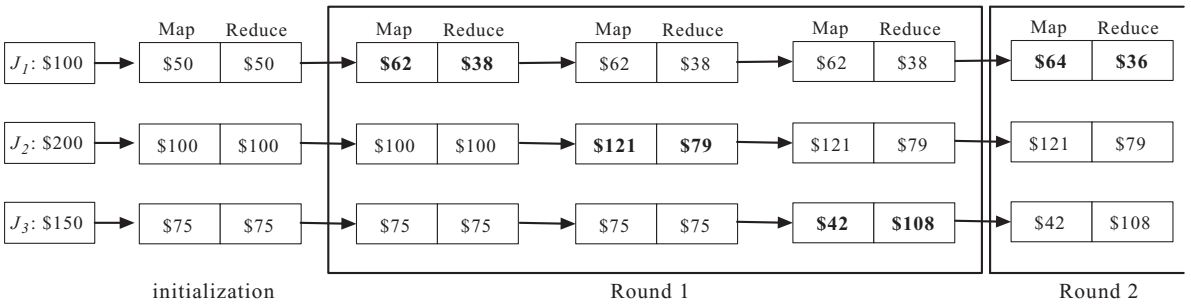


Fig. 4. Auctioneer automatically optimizes the sub-budgets of all jobs in order, until no job can further improve.

function as well as Lemma 1. Specifically, it reassigns the sub-budget b'_{ij} for job J_i, proportional to $\frac{\partial u_i}{\partial s_{ij}} s_{ij}(1 - s_{ij})$. The following lemma shows $\{b_{ij}\}$ is the optimal sub-budget maximizing the utility if the sub-budget configuration reaches a fixed point.

Lemma 2. *In Algorithm 2, $b_{ij} = b'_{ij}$ for all j, only if $(b_{i1}, \ldots, b_{im})$ is the optimal sub-budget combination maximizing J_i's utility.*

In Fig. 4, we elaborate the optimization procedure with our running example. In the initial configuration, all jobs evenly partition the budget on map and reduce resources. In the first round of optimization iterations, J_1 finds the sub-budgets ($62, $38) on the resources. In the following, jobs J_2 and J_3 optimize their sub-budgets in order. J_1 is allowed to optimize again when the first round of optimization is done. In the new iteration, J_1 slightly changes his sub-budgets. The optimization quickly converges when no job can further improve their utility by shifting the budgets among resources. In the theorem below, we theoretically prove the convergence property of the algorithm.

Theorem 1. *Algorithm 1 always terminates after a finite number of iterations, if α is no larger than* 0.5.

While the theorem is valid only when α is no larger than 0.5, we tested α values larger than 0.5 and find the convergence is always reached. Moreover, although the theorem does not characterize the convergence rate of the computation process, experimental result in the rest of the paper also verify the fast convergence in terms of the number of iterations. We believe the convergence rate depends not only on the value of α but also on the strong convexity property of the utility function. Based on the proof of Theorem 1, it is obvious that smaller α leads to more significant update on the assignment matrix, and probably faster converge with fewer iteration. These properties are fully evaluated in the experiments, while deeper theoretical investigation are left as future work.

4.3. Economical features

There are some attractive economical features in Abacus, which guarantees that all users fully trust the system and they always prefer to submit their true job profiles. In particular, the mechanism satisfies two properties. First, it is unnecessary for the users to worry about the optimality of the sub-budgets the system calculates for his jobs. It is impossible to gain additional utility for users by submitting sub-budgets for the jobs. Second, the mechanism is *incentive compatible*. It is a dominating strategy for the users to tell the maximal budget he is willing to pay and the true utility function on his jobs. These properties are proved by the following lemmata and theorems.

Lemma 3. *The user cannot achieve higher utility if he is allowed to modify sub-budgets by himself.*

Therefore, it becomes meaningless for the users to submit the sub-budgets based on their own understandings. If this property is not satisfied, the users will be willing to control the sub-budgets by themselves. This potentially leads to trials on different sub-budget strategies and hurts the system efficiency when every user is testing and changing their strategies continuously. A similar phenomenon has been observed in sponsored search market, when the first price auction mechanism was employed a decade ago [18].

Lemma 4. *The utility of a job J_i is monotonic increasing if the user increases its budget b_i on the job.*

Theorem 2. *If every user has at most one running job at any time, the auction mechanism used in Abacus is incentive-compatible.*

The theorem implies that all users will definitely tell the "truth" about their budget and utility function. Recall our running example in Fig. 2. It means that the owner of job J_1 maximizes the utility of J_1 by submitting current profile. The utility of the job can never be larger if he fakes a wrong utility function or tells a lower budget of the job. When every user has multiple running jobs, however, it remains possible for the user to manipulate the job profiles to achieve higher overall utilities on the jobs.

5. Utility function estimation

In practice, it can be difficult for an ordinary user of the cloud system to come up with the exact utility function of her/his jobs. To alleviate the problem, in this section we propose a novel approach that automatically estimates the utility function of a user's job, given statistics of similar jobs run by the same user in the past. This enables the user to participate in the auction with only her/his budget information. Our estimation is purely based on analysis over the

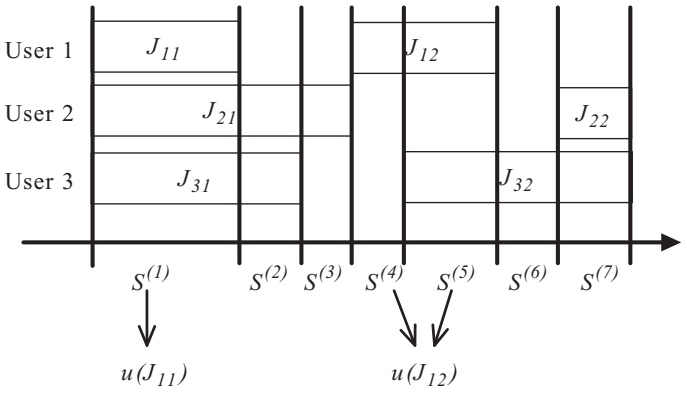


Fig. 5. An example of running jobs with three users, in which u_1's utility function is possibly wrong.

utilities/performance of the jobs running with utility functions assigned by the system. Since the non-decreasing concave function $g_j(s_{ij})$ can be of arbitrary form, we focus on two common classes of functions introduced in Section 3, namely, *linear utility functions* and *logscale utility functions*. Functions in either of the classes are uniquely controlled by d weight coefficients $\{w_1, w_2, \ldots, w_d\}$, i.e. $u_i(s_i) = \sum_{j=1}^{m} w_j s_{ij}$ and $u_i(s_i) = \sum_{j=1}^{m} w_j \log s_{ij}$.

Generally speaking, *Abacus* allows the user to submit the budget only, without the utility function. In such situations, the system automatically completes the bid by assigning a default utility function, e.g. linear utility function with equal weights. The system thus calculates the assignment probabilities based on the bid with the actual budget but possibly wrong utility function. After a few rounds of job or service running, when sufficient statistics on the performance of the jobs are collected, *Abacus* is capable of deriving the exact utility function.

In Fig. 5, we present an example of the running jobs on two types of resources with three users. Assume that user 1 does not know his exact utility function and the system assign the utility function $u_1(x, y) = x + y$. At the beginning, job J_{11} is started when J_{21} and J_{22} are already running. The resource allocation matrix $S^{(1)}$ is calculated based on bids from all three users. With the evolving of the system, different job combinations lead to different resource allocations. It is clear that the job J_{11} is completed in the system when $S^{(1)}$ is used as the only resource setting during the whole procedure. Job J_{12} is run in the system under two different settings, i.e. $S^{(4)}$ and $S^{(5)}$. On the other hand, the system records the performance of J_{11} and J_{12} and transform to the measurable utility, i.e. $u(J_{11})$ and $u(J_{12})$. If the jobs are Map-Reduce processing, for example, the utility could be the inverse of the job response time. If the jobs are web services, as another example, the utility could be the average throughput of the service server. Given the utility measures, the system estimates the utility functions via a system of linear equations:

$$u(J_{11}) = w_{11}s_{11}^{(1)} + w_{12}s_{12}^{(1)},$$
$$u(J_{12}) = w_{11}\left(\gamma s_{11}^{(2)} + (1-\gamma)s_{11}^{(3)}\right) + w_{12}\left(\gamma s_{12}^{(2)} + (1-\gamma)s_{12}^{(3)}\right).$$

In the equation above, γ denotes the ratio of $S^{(4)}$ takes in the running time of job J_{12} in the system. Thus, $\gamma s_{11}^{(2)} + (1-\gamma)s_{11}^{(3)}$ denotes the average amount of resource type 1 which job J_{12} is expected to get. Since there are only two unknown variables in the linear system, we are able to recover the exact values, by calculating the unique solution to w_{11} and w_{12}. This means that utility measures on m jobs are sufficient to reconstruct the (linear or logscale) utility functions, in which m is the number of resource types. However, due to the performance fluctuation commonly observed in cloud system, this scheme may not return robust estimation of utility function all the time. To improve the robustness of the estimation method, we formalize a regression-based technique as follows.

Assume that *Abacus* collects the utility measures on k ($k \geq m$) repeated jobs from a particular user. Without loss of generality, let $\{J_{i1}, \ldots, J_{ik}\}$ denote the jobs. We use *Average Job Resource* to estimate the resource allocated to the jobs when running in *Abacus* system.

Definition 1. Average job resource. Regarding a job J_{il} from user U_i, the *Average Job Resource* is a vector of length m, i.e. $\bar{s}_{il} = (\bar{s}_{il1}, \ldots, \bar{s}_{ilm})$, such that $\bar{s}_{ilj}$ is the average probability of assigning resource j to job J_{i1} from the beginning to the completion of the job.

Based on the definitions above, the input to the utility function estimation component includes (1) the utility measures $\{u(J_{i1}, \ldots, u(J_{ik})\}$; and (2) the average job resource of the jobs $\{\bar{s}_{i1}, \ldots, \bar{s}_{ik}\}$. Given such statistic information, we run a regression to find out the optimal weights to minimize the following objective function.

$$\text{Minimize: } \sum_{l=1}^{k}\left(u(J_{il}) - \sum_{r=1}^{m} w_{il}\bar{s}_{il}\right)^2 \tag{2}$$

It is straightforward to verify that the regression formulation is simply linear, which can be easily solved by running standard solution to linear regression. The computation time is cubic, i.e. $O(m^3)$, in terms of the number of resource types. Since m is usually not large in real cloud system, the overhead of the utility function estimation is affordable.

6. SLO-based scheduling in Abacus

So far, we have assumed that the user's utility for a job can be expressed as a concave function of the allocated resources, regardless of the amount of resources given to the job. In practice, certain types of jobs must be finished within a given timeframe. This means that insufficient resources lead to violation of the deadline, and, consequently, zero utility for the job. Such constraints are usually expressed by *service level objectives* (*SLOs*), which are commonly used in cloud services. Although an SLO can be incorporated into the job's utility function (i.e., the utility remains zero until there are sufficient resources to meet the SLO), such a utility function is no longer concave, which cannot guarantee the convergence and truth-telling properties of the Abacus auction. In this section, we propose an effective scheduling scheme to enforce SLOs within the Abacus framework, while preserving the desirable properties of the Abacus auction described in Section 4.3.

We model SLOs as constraints. Specifically, the user only pays the cloud service for running a job when the job's SLO is met. The SLO for a job specifies the minimum utility for the job. Formally, a job is now expressed as $J_i = (b_i, u_i, m_i)$, in which b_i and u_i follow the original definitions in Section 3, and m_i indicates the minimal utility requirement specified in the SLO.

The main idea of the proposed solution is to postpone jobs that cannot possibly meet their SLOs until there are unused system resources. The solution relies on a job utility estimation module, which is capable of returning an accurate estimation on the expected actual utility of a job. For MapReduce jobs, for example, ARIA [37] provides an algorithm outputting completion estimations on the jobs. Mathematically, given any group of jobs $\mathbb{J} = \{J_1, J_2, \ldots, J_n\}$, the job utility estimation module returns the utility assignment $\{u_1, u_2, \ldots, u_n\}$ based on the current availability of resources.

Based on this module, we design a variant of Abacus, called *SLO-based Abacus*, to maximize the profit of the system, while preserving all desirable economic properties of Abacus. There are two steps in SLO-based Abacus. First, we adopt a greedy strategy to find the most prolific job group such that current cloud computation resources are sufficient to meet their SLOs. In this way, we divide all running jobs into two sets, *Prolific Pool* and *Non-Prolific Pool*. Second, the framework assigns resources to the jobs in prolific pool by running the standard auction-based Abacus algorithm. If there are unused resources available, the framework randomly picks up a job from non-prolific pool to use the resource. In the following, we prove two important properties of SLO-based Abacus, which ensures that no user is willing to submit a job with a false budget or SLO.

Lemma 5. *A rational user never submits job J_i with incorrect budget $b_i' \neq b_i$.*

Lemma 6. *A rational user never submits job J_i with incorrect minimal utility $m_i' \neq m_i$.*

As a conclusion of the theoretical analysis, SLO-based Abacus preserves the truth-telling properties of the original Abacus framework. It is thus applicable to cloud systems processing jobs with minimal utility SLOs. In the experimental section, we will discuss how SLO-based Abacus is used in Hadoop to handle MapReduce jobs with SLOs.

7. Experiments

In this section, we conduct empirical studies on the performance and properties of *Abacus* in a real cloud system. Section 7.1 reports results of experiments run on a private cloud system, and Section 7.2 presents results of experiments run on public cloud, i.e. *Amazon EC2*.

Abacus on Hadoop: to test the usefulness of *Abacus* on practical cloud systems, we added a new auction component into Hadoop 0.20, by redesigning the scheduler in Hadoop based on *Fair Scheduler* [40]. Basically, our meta scheduler monitors the profiles as well as the submitted budgets of the active users. We regard *Map* and *Reduce* nodes as two independent types of resources. When an event of job arrival or departure happens, the auction component recalculates the allocation probabilities for each active job on map nodes and reduce nodes, using our auction mechanism. The probabilities are fed into the low-level Job Pool scheduler [40] to ensure effective resource allocation and scheduling. The source codes of the implementation and installation instructions on Hadoop 0.20 system are available online.[1]

7.1. Experiments on private cluster

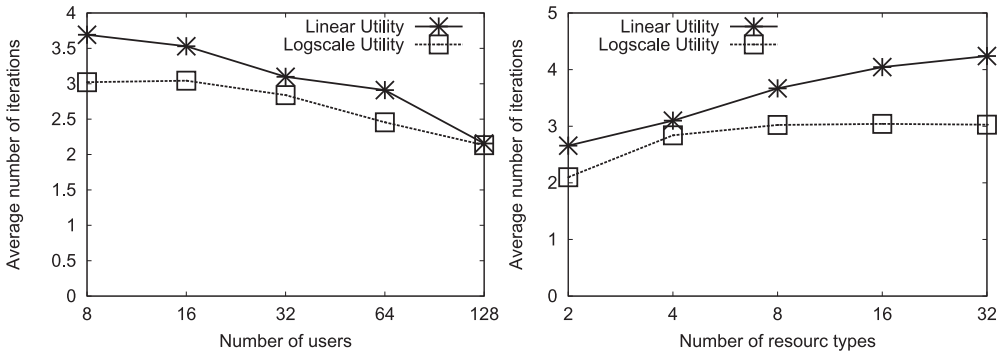
We test Abacus on the *Epic* platform,[2] a distributed computation cluster deployed at the National University of Singapore. Epic consists of 72 computing nodes. The master node is equipped with a dual-core 2.4 GHz CPU, 48 GB RAM, two 146 GB SAS hard disks and another two 500 GB SAS hard disks. Each slave node has a single-core 2.4 GHz CPU, 8 GB RAM and a 500 GB SATA hard disk. All the nodes used in the cluster are running CentOS 5.5 and Hadoop 0.20.

7.1.1. Auction efficiency

We first evaluate the auction component. In the experiments, we measure (1) the number of iterations before convergence, given the bids from all of the users; and (2) the total computation time for the resource allocation matrix. To manually control the setup of the bid combination, experiments in this subsection are *not* run on the Hadoop system. Instead, we independently test the auction component without running the resource allocation on real computation tasks. We repeat each experiment 1,000 times, and report the average measurements as well as 95% quantiles. The default setting of the experiments involves 32 system users, 4 types of resources, and a balancing parameter $\alpha = 0.5$. The budgets of the users follow an independent and identical uniform distribution in the range [50, 200]. Both *linear utility* function and *logscale utility* function are tested. Every weight w_j on resource j in the utility functions follows a uniform distribution on range [0.5, 2].

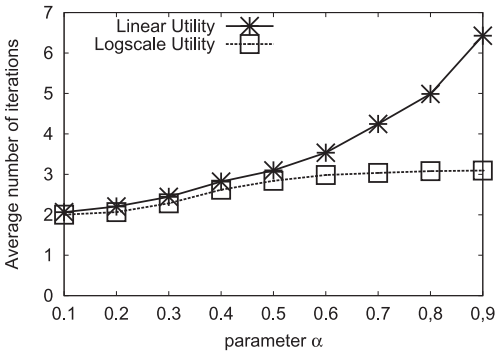
Fig. 6 shows the average number of iterations of the auction algorithm under different settings. The auction algorithm generally converges very quickly, within 5 iterations in

[1] https://sites.google.com/site/zhangzhenjie/abacus.zip .
[2] http://www.comp.nus.edu.sg/~epic .



(a) No. of users vs. # of iterations

(b) No. of resource types vs. # of iterations



(c) α vs. # of iterations

Fig. 6. Experiments on # of iterations of the auction algorithm.

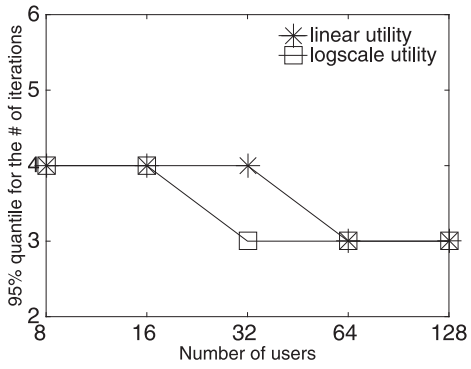
most settings. When there are more users in the auction, as is shown in Fig. 6(a), the convergence rate is accelerated. This is because every user possesses a small fraction of resources in the system, which is not dramatically affected by the iterations. Therefore, the resource allocation quickly reaches a Nash Equilibrium after a few rounds of updates on the resource allocation. When increasing the number of resource types (Fig. 6(b)), the convergence slightly slows down, since it potentially takes more iterations for a specific user to move budgets from certain resources to others before arriving at the optimal portfolio. In Fig. 6(c), we show that the impact of the balancing parameter α is very different on the linear utility and logscale utility functions. For logscale utility functions, the convergence rate remains constant regardless of α. For linear utility functions, the number of iterations almost doubles, when α changes from 0.5 to 0.9. The reason is that the linear utility function tends to give all resources to the user with highest budget, when α is close to 1. This leads to more iterations to gradually move resources towards a few top-tier users with highest budgets. Logscale utility function, on the other hand, is more robust against α, leading to fewer iterations with large α.

Fig. 7 shows the 95% quantile results for the above experiments; for example, a 95% quantile of 4 means that among the 1,000 runs of the algorithm, 950 (i.e., 95%) runs terminated within 4 iterations. The results exhibit similar trends as the average results shown in Fig. 6. More importantly, observe that the 95% results are close to their corresponding average values, meaning that the auction algorithm consistently converges within a few iterations.

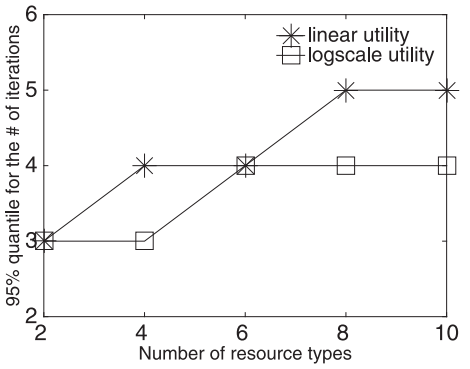
Fig. 8 evaluates the total computation time of the auction. When increasing the number of active users (Fig. 8(a)) or the number of resource types (Fig. 8(b)), the computation time also increases. The growing computational cost is mainly due to the increasing CPU time on each iteration, which needs to redistribute the probabilities on every pair of user and resource type. However, even when there are 128 active users submitting bids to the auction component, the auction finishes within 7 ms by average, which is negligible. This implies that the proposed auction algorithm can handle a large number of job arrivals or departures.

7.1.2. *Incentive compatibility*

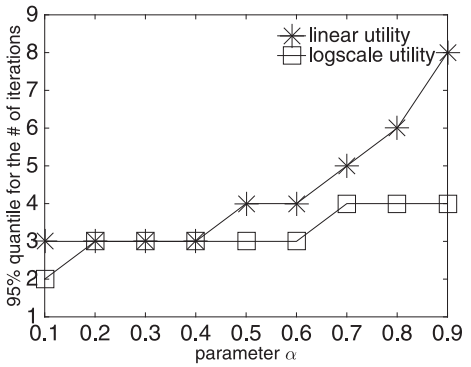
Next we verify the incentive compatibility property of *Abacus* on real systems. To do this, we simulated three different users that submit different types of MapReduce jobs. When generating the workload, we assumed that job arrivals of the three users follow three independent Poisson processes with different arrival rates. In order to create jobs that require different utility functions, we carefully controlled the running time of the Map and Reduce tasks. In Table 1, we list the average execution time for the Map and Reduce tasks of the three users. During the simulation, the users keep submitting new jobs to the system for 30 min from the beginning of the experiments. The system stops after finishing all submitted jobs. We use the inverse of job response time as



(a) No. of users vs. # of iterations

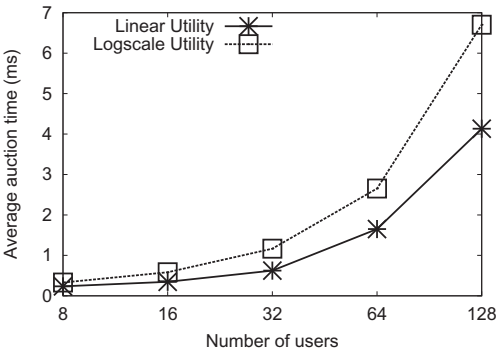


(b) No. of resource types vs. # of iterations

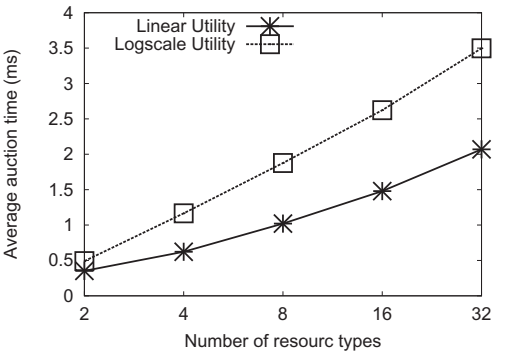


(c) α vs. # of iterations

Fig. 7. Experiments on 95% quantile for the number of iterations of the auction algorithm.



(a) varying # of users



(b) varying # of resource types

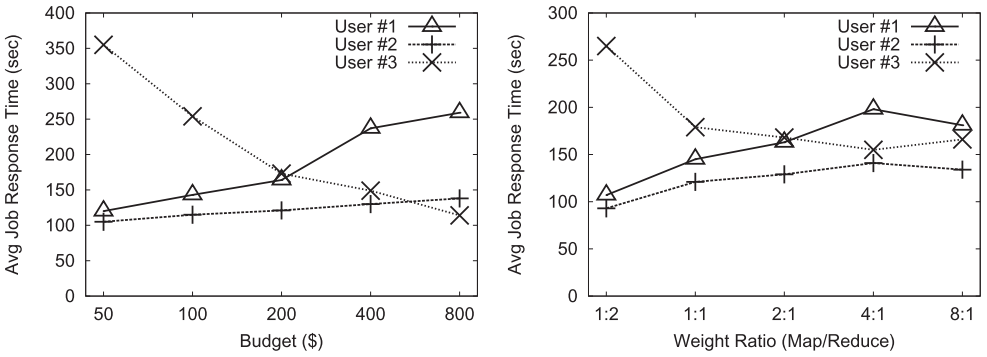
Fig. 8. Experiments on the computation time of auction algorithm.

Table 1
Settings on the users and jobs.

User	Map	Reduce	Budget	Arrival rate
User #1	30 sec.	20 sec.	$200	$\lambda = 2$/min
User #2	15 sec.	40 sec.	$200	$\lambda = 1.5$/min
User #3	60 sec.	10 sec.	$200	$\lambda = 1.1$/min

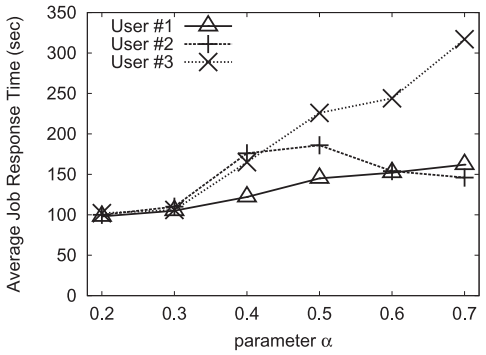
the measure for job utility. The actual utility function is thus proportional to the processing demands on Map and Reduce tasks. Therefore, we simply employ linear utility function $u(x, y) = T_m x + T_r y$ in our simulations, in which T_m and T_r are proportional to the expected running time for each Map and Reduce task respectively. For user #1, for example, the utility function is $u_1(x, y) = 3x + 2y$. Although this simple linear model does not fully reflect the coupling relationship between Map and Reduce tasks [33,34], our experimental results imply that it is sufficient to model the job performance when most of the jobs in the system are *short* jobs.

The most important implication of incentive compatibility is that the users maximize their job efficiency by submitting the maximum affordable budgets and true utility



(a) Budget of user #3 vs. response time

(b) Utility Function of user #3 vs. response time



(c) α vs. response time

Fig. 9. Experiments on the incentive compatibility property.

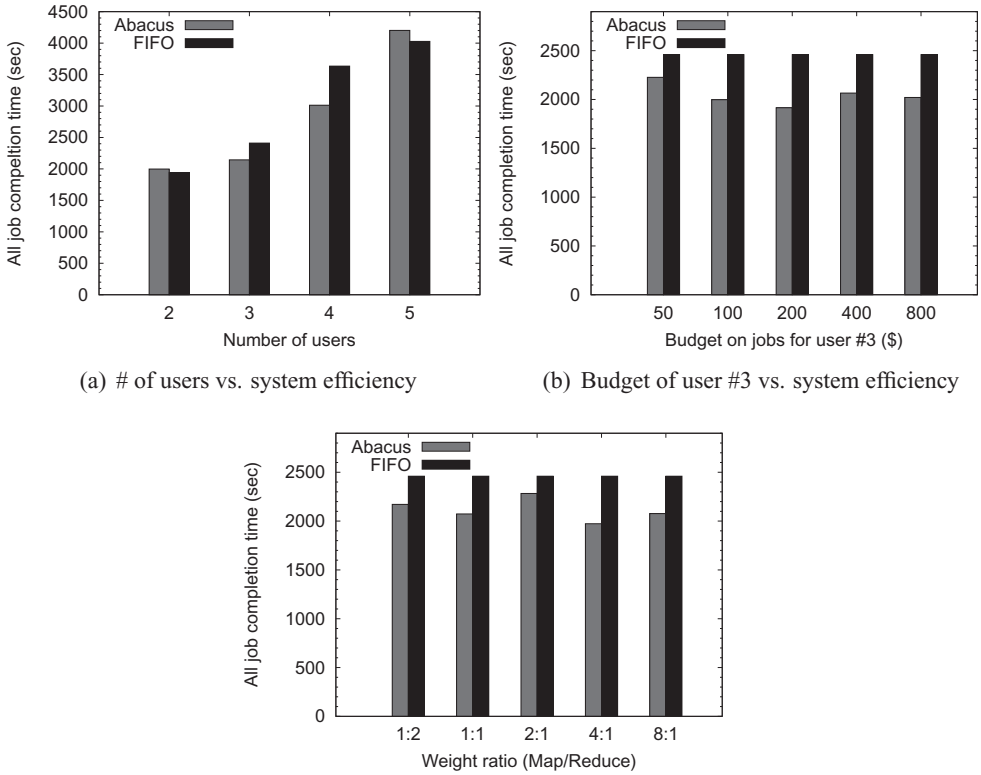
functions. To show this, we vary the budget for user #3 and fix the budgets of the other two users. When the budget of user #3 grows, the efficiency of the jobs from user #3 improves, as shown in Fig. 9(a). The efficiency gains slow down as the amount of budget increases, due to the concavity of the utility function. On the other hand, the performance of the jobs from the other two users decreases, since their bids are weakened when another participant spends more.

Since all jobs employ linear utility functions in our setting, we vary the weights of the utility function u_3 for user #3 to evaluate the impact of truthfulness of the utility functions (Fig. 9(b)). Our test covers 5 different ratios of the map to reduce weight. When the ratio is 4:1, for example, the corresponding utility function is $u_3(x, y) = 4x + y$. As shown in Table 1, the jobs from user #3 take about 60 s on maps and 10 s on reduces. Thus, the true ratio of map to reduce weight is about 6:1. In Fig. 9(b), we present the results on the job efficiency for all three users. The job efficiency for user #3 is maximized when the ratio on weights is 4:1, which is closest to the true ratio of Map running time to Reduce running time. Since the total computation resource is constant, the jobs from other users taking longer to finish when user #3 is reporting his true utility.

In Fig. 9(c), we test the impact of the balancing parameter α on job efficiency. With a small α, Abacus assigns resources to all jobs with almost equal probability, leading to almost identical average job processing time. With larger values of α, Abacus tends to distinguish jobs based on their profiles.

7.1.3. System efficiency

We next focus on the overall system performance instead of individual job response time. We compare Abacus against the popular First-In-First-Out (FIFO) scheduling strategy, and report the total time to finish all jobs. We aim to show that Abacus enables service differentiation with small overhead on the overall performance of the system. Fig. 10 shows that Abacus has competitive performance compared to FIFO, with a varying number of users in the resource auction. Note that in this set of experiments, all system parameters are fixed except for the number of users; in particular, the total amount of computational resources is constant in all experiments. According to the results, when the number of users is relatively low (i.e., less than 5), Abacus outperforms FIFO on system overall performance in most settings, due to the clear statement on the utility functions on the jobs: the system is capable of better scheduling the assignment of map and reduce resources to improve overall throughput. When we increase the number of users beyond a certain point (5 in our experiments), resource contention increases with the number of users, which affects the relative performance of Abacus and FIFO. In particular, Abacus involves more context switching than FIFO, since the former needs to balance jobs from different users based on their bids and utility functions, whereas the latter simply executes each incoming job with all available computational resources. Nevertheless, the total completion time of Abacus in these settings is only slightly worse (at most 5% more) compared to FIFO.



(a) # of users vs. system efficiency

(b) Budget of user #3 vs. system efficiency

(c) Util. func. of user #3 vs. system efficiency

Fig. 10. Experiments on system efficiency.

In Fig. 10(b), an interesting observation is that the system performs well when all jobs have the same budget, i.e. when user #3's budget is $100. This is because Abacus works similarly to *Fair Scheduler* when all users have the same budget/priority. In this case, the system assigns the resources based on the preference information contained in the utility functions, instead of purely based on their budgets.

We also tested the system performance when the users are submitting different job utility functions. Fig. 10(c) shows the impact of user #3 submitting different utility functions. The results again confirm the superiority of Abacus over FIFO. Abacus improves the system performance by about 20%, even when the user is not reporting the true utility functions. The performance gain increases further when the user tells her true utility function.

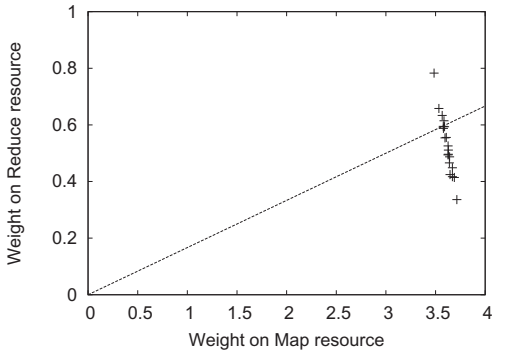
7.1.4. Utility function estimation

Finally, we evaluate the effectiveness of the utility function estimation technique described in Section 5. To understand the job utility for a particular user, we record the utilities (i.e., inverse of the running time) of the MapReduce jobs from user #3 in the Hadoop system. These jobs are run by setting different utility functions, as is done in the experiments in Section 7.1.2. The implementation of the estimation algorithm is not included in the published source code package (since it requires non-free libraries), and it is available upon request.

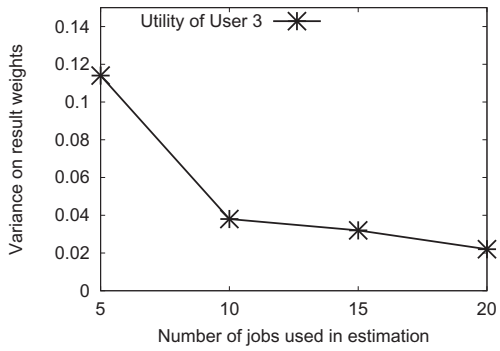
In Fig. 11(a), we plot 20 result utility functions by mapping each function to a 2D point. Each cross denotes a utility function, calculated by running our algorithm with utility measures on 5 jobs from user #3. The coordinates of each cross on each dimension represents the weight of the function on map/the weight on reduce. Another line plotted in the figure represents the true utility function of the user. Each point on the line indicates a utility function, in which the weight for map is 6 times as that for reduce. The functions on the line thus match the actual requirements of the jobs from user #3. As shown in the figure, the estimated utility functions are generally close to the true utility function. The ratios of the weight on map/reduce weight ratio usually falls in the range from 5 to 9 (the true value being 6). Fig. 11(b) reports the variance of the result when varying the number of jobs. Clearly, the variance plunges as the number of jobs increases. Fig. 11(c) shows that all utility function estimations finish within 3 ms, which is negligible.

7.2. Experiments on Amazon EC2

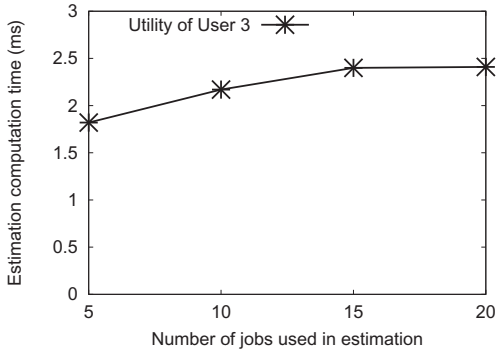
The experiments in previous subsection are completely run on a private cluster. In this part of the section, we test Abacus in Hadoop on a public cloud system, i.e. Amazon EC2. Since EC2 are purely based on virtualization techniques, the performance of the VMs are less stable than the machines in the private cluster. In particular, we aim to evaluate the *Incentive Compatibility* and *System Efficiency* of Abacus on



(a) Distribution of the result utility function



(b) Variance of the result utility function



(c) Computation time of the estimation

Fig. 11. Experiments on utility function estimation.

such public cloud with large variety on computation capacity. Note that *Auction Efficiency* and *Utility Function Estimation* are not reported in this subsection, since they do not depend on the infrastructure of the underlying cloud system. In particular, we emphasize that the auction algorithm is independently of the underlying cloud platform, and, thus, the convergence and efficiency results presented in Section 7.1.1 also applies to settings in this subsection.

In our experiments on EC2, we rent 15 virtual machines under the category of *M1.large*. Without otherwise specification, we use all these virtual machines to run our modified version of Hadoop system. The workload we use is different from the jobs introduced in previous subsection. In particular, we follow the workload construction instructions in [34], with slight changes. In Table 2, we list 10 types of MapReduce jobs run by the users in our experiments. The columns of *Map* and *Reduce* indicate the number of map tasks and reduces tasks for particular jobs. The table shows that Job #6 and Job #9 are reduce-intensive jobs and all the others are map-intensive jobs. The Wikipedia dataset is downloaded from http://wiki.dbpedia.org/Downloads38.

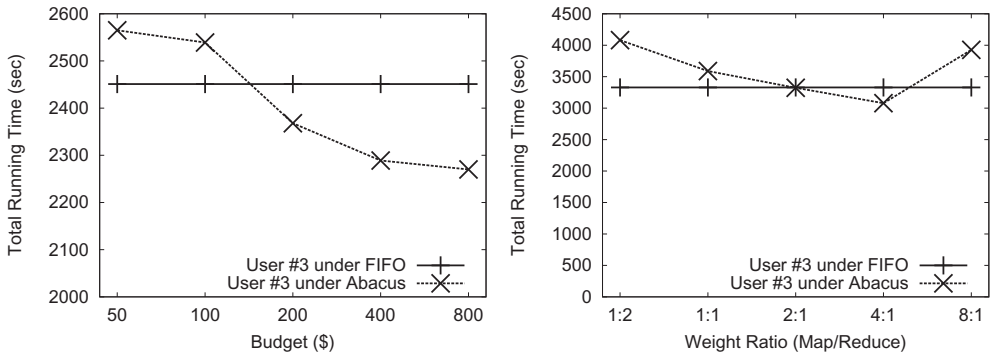
Table 2
Workload of MapReduce jobs run on Amazon EC2.

ID	Description	Map #	Reduce #
1	Grep [a–g][a–z]∗ wikiInput	427	14
2	Grep [1–2]∗ random05	8	2
3	WordCount wikiInput	15	1
4	QuasiMonteCarlo	10	1
5	Grep [1–2]∗ random10	15	2
6	Sort randomPair2	352	14
7	Grep [1–4]∗ random05	8	2
8	Grep [1–2]∗ random10	15	2
9	Sort randomPair1	224	14
10	Grep [1–5]∗ random05	8	2

7.2.1. Incentive compatibility

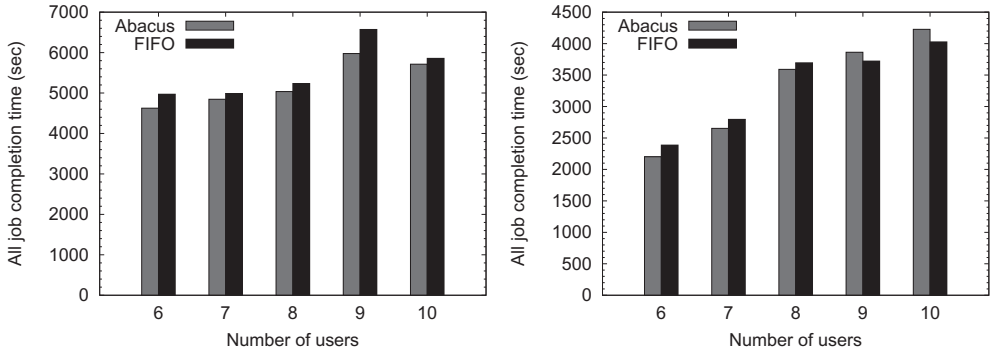
To generate accurate linear utility functions, we first test the running time of the map and reduce tasks of all jobs. Recall that each of 10 users is associated with a job in the experiments. Therefore, based on the running time estimations, the linear utility function for a user U_i is constructed as $u_i(x, y) = T_m x + T_r y$, in which w_x and w_y are the total running time of all map and reduce tasks under the job J_i. Moreover, the default budget for each user is \$200. Given the utility functions and budgets assigned to the users, we verify the property of incentive compatibility of the Abacus system.

Similar to the experiments on private cluster, we vary the budget of user U_3 from \$50 to \$800, as is shown in Fig. 12(a). We compare the performance of the jobs under Abacus against the performance of the jobs under FIFO. Since all the other users are willing to pay \$200 for their jobs, the running time of the jobs under Abacus are significantly higher than that of the jobs under FIFO, when the budget is below \$200. A quick reduction on the running time is observed when the budget is raised from \$100 to \$400.



(a) varying the budget of user #3 (b) varying the utility of user #3

Fig. 12. Experiments on the job efficiency of user #3 when varying budget and utility function.



(a) increasing # of users and VMs at the same time (b) increasing # of users only

Fig. 13. Experiments on the system efficiency of Abacus when varying # of users.

However, the benefit is no longer significant, if we further increase the budget of user U_3, mainly due to the marginalization property of the utility function.

The second group of experiment is conducted to test the impact of the utility function. From the result in Fig. 12(b), it is clear that the running time of the jobs from U_3 is minimized when the weight ratio T_m/T_r is 4, which is closest to the actual ratio of the total running times of map and reduce tasks of the jobs. It proves that our mechanism encourages the users to submit the bids with actual needs on the resources.

7.2.2. System efficiency

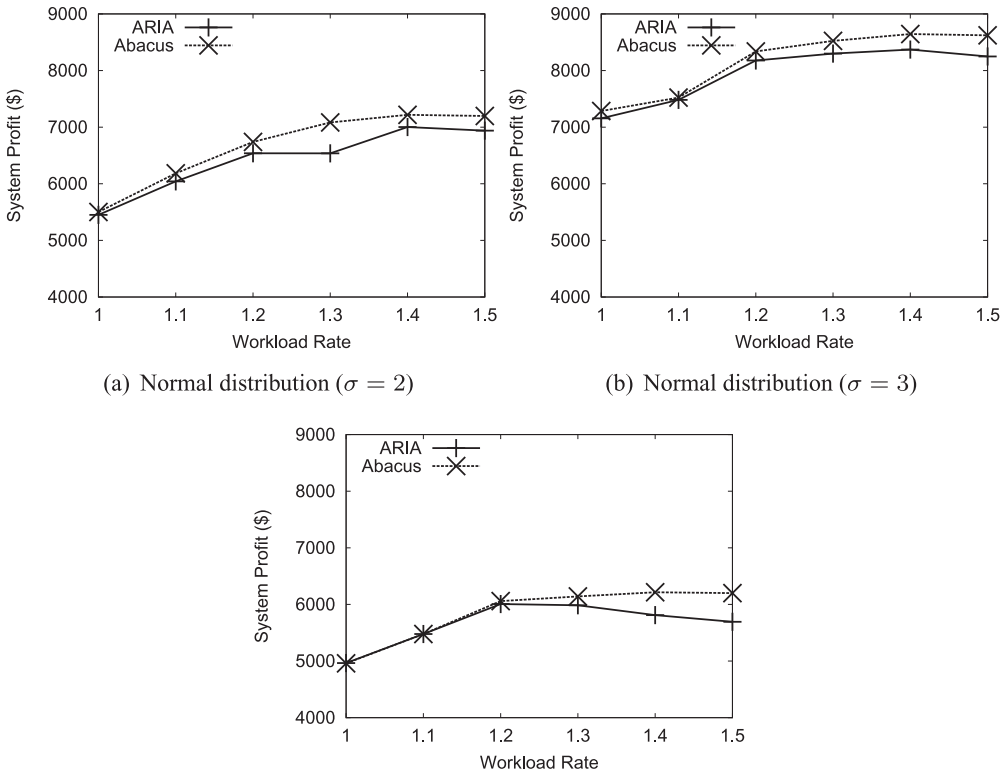
Finally, we evaluate the overall system efficiency by reporting the total running time of all jobs from all users. FIFO is the baseline approach we compare against in this part of the subsection.

These experiments are divided into two groups. In the first group of the experiments, we vary the number of users in the system and the number of virtual machines rented on EC2 at the same time. Therefore, the computational resource increases in proportional to the number of users. Fig. 13(a) shows that Abacus consistently outperforms FIFO, as long as the average resource among all users remains almost the same. It implies that Abacus is an ideal mechanism balancing between fairness and priority.

The second group of the experiments are done in a very different way. We use all 15 virtual machines under all settings with different number of users. The results are presented in Fig. 13(b). When there are less users in the system, the competition over the resource is not high. Abacus is competitive against FIFO, with about 5% margin advantage on the total running time. However, when there are too many users in the system competing for computation resource, the overall running time of the jobs in Abacus becomes longer than that under FIFO. This is basically due to the transition cost incurred by the scheduling method used in Abacus. Particularly, when a high-priority job comes into the system, existing jobs must give up the resource before completing the MapReduce job. Such results show that priority leads to unfair resource allocation and affects the overall performance when the amount of the resource is seriously below the demands from the users.

7.2.3. SLO-based scheduling

In Fig. 14, we present the system profits of Abacus and ARIA [37], which is the state-of-the-art solution to Hadoop scheduling on jobs with service level objectives. We test both algorithms with random deadlines assigned to the MapReduce jobs tested in our previous experiments, where the random deadlines follow three different settings: a normal distribution with variance 2, a normal distribution with variance 3 and a uniform distribution in the range [500, 1000]. To



(a) Normal distribution ($\sigma = 2$)

(b) Normal distribution ($\sigma = 3$)

(c) Uniform distribution

Fig. 14. System profit vs. deadline distribution in SLO-based scheduling.

vary the workload of the system, we manipulate the number of MapReduce jobs running in the system. A workload with rate r contains jobs requiring r times of the computation capacity of the current system. It thus leads to more resource competition when the workload rate increases. The results show that Abacus obtains competitive overall profit of the system compared to ARIA. Note that besides overall profit, Abacus also achieves service differentiation with truthfulness and fast-converging auctions, whereas ARIA provides none of these features.

8. Conclusion and future work

We present Abacus, a new auction-based resource allocation framework for cloud systems. Abacus provides effective service differentiation for jobs with different budgets, utility properties and priorities. We design a novel auction mechanism that is incentive-compatible and highly efficient. Experiments on a real cluster show promising performance. Abacus currently only handles independent computation resources in the system. Hence, an interesting direction for future work is to handle dependent resources, e.g., using a dependency model. We prove that the auction algorithm in Abacus always converges to a Nash Equilibrium, and experiments show that the algorithm usually converges in a small number of iterations. Hence, another interesting topic for further study is to theoretically analyze its convergence speed.

It is more challenging to decouple the dependent resources during resource allocation, while preserving efficiency and fairness of the cloud system. It makes it difficult to design resource allocation mechanism for complicated cloud applications, such as distributed streaming processing systems [42]. We aim to overcome these problems with new dependency models between the resources.

Acknowledgments

Zhang, Ma and Yang are supported by the HCCS Program from Singapore's A*STAR. Ma is also supported by AcRF grant 1054R-252-000-448-133 from Singapore's Ministry of Education. Ding is partially supported by the NSF of China under Grant 61173081, the Guangdong Natural Science Foundation, China, under Grant S2011020001215, and the Guangzhou Science and Technology Program, China, under Grant 201510010165. The authors would like to thank Prof. Hongyang Chao for her support.

Appendix

Lemma 1

Proof. First of all, based on chain rule of derivative, we have $\frac{\partial u_i}{\partial b_{ij}} = \frac{\partial u_i}{\partial s_{ij}} \cdot \frac{\partial s_{ij}}{\partial b_{ij}}$. Since $u_i(s_i) = \sum g_j(s_{ij})$ and g_j is non-decreasing concave function, it is straightforward to verify that $\frac{\partial u_i}{\partial s_{ij}} > 0$ and $\frac{\partial u_i}{\partial s_{ij}}$ decreases monotonically with s_{ij}. On the

other hand, Eq. (1) implies that $\frac{\partial s_{ij}}{\partial b_{ij}}$ can be reformulated as follows.

$$\frac{\partial s_{ij}}{\partial b_{ij}} = \frac{\alpha (b_{ij})^{\alpha-1}\left(\sum_l (b_{lj})^{\alpha}\right) - \alpha (b_{ij})^{\alpha-1}(b_{ij})^{\alpha}}{\left(\sum_l (b_{lj})^{\alpha}\right)^2} = \frac{\alpha (b_{ij})^{\alpha-1}\sum_{k\neq i}\left((b_{kj})^{\alpha}\right)}{\left(\sum_l (b_{lj})^{\alpha}\right)^2}$$

When $0 \le \alpha \le 1$, $(\Sigma_l (b_{lj})^{\alpha})^2$ is monotonic increasing and $(b_{ij})^{\alpha-1}$ is monotonic decreasing, with respect to b_{ij}. Therefore, we can prove the lemma, since both $\frac{\partial u_i}{\partial s_{ij}}$ and $\frac{\partial s_{ij}}{\partial b_{ij}}$ are non-negative monotonic decreasing functions. □

Lemma 2

Proof. Based on the optimality condition, a sub-budget combination $\{b_{i1}, b_{i2}, \ldots, b_{im}\}$ is optimal if and only if (1) $\sum_j b_{ij} = b_i$; and (2) we can find some positive constant C such that $C = \frac{\partial u_i}{\partial b_{i1}} = \frac{\partial u_i}{\partial b_{i2}} = \cdots = \frac{\partial u_i}{\partial b_{im}}$.

When $b_{ij} = b'_{ij}$ for every j, the first condition is definitely satisfied. In the following, we prove $\{b_{ij}\}$ are also consistent with the second condition. First of all, the partial derivative $\frac{\partial u_i}{\partial b_{ij}}$ can be rewritten as $\frac{\partial u_i}{\partial b_{ij}} = \frac{\partial u_i}{\partial s_{ij}} \cdot \frac{\partial s_{ij}}{\partial b_{ij}} = \frac{\partial u_i}{\partial s_{ij}} \frac{\alpha}{b_{ij}} s_{ij}(1 - s_{ij})$.

On the other hand, Algorithm 2 calculates b'_{ij} as follows:

$$b_{ij} = b'_{ij} = \frac{\frac{\partial u_i}{\partial s_{ij}} s_{ij}(1 - s_{ij})}{\sum_{k=1}^{n}\left(\frac{\partial u_k}{\partial s_{kj}} s_{kj}(1 - s_{kj})\right)} b_i$$

Therefore, we can further derive as follows:

$$\frac{\partial u_i}{\partial b_{ij}} = \frac{\partial u_i}{\partial s_{ij}} \frac{\alpha}{b_{ij}} s_{ij}(1 - s_{ij}) = \frac{\alpha}{b_{ij}} \cdot \frac{b_{ij}}{b_i} \sum_{k=1}^{n}\left(\frac{\partial u_k}{\partial s_{kj}} s_{kj}(1 - s_{kj})\right) = \frac{\alpha}{b_i} \sum_{k=1}^{n}\left(\frac{\partial u_k}{\partial s_{kj}} s_{kj}(1 - s_{kj})\right) \quad (A.1)$$

This shows that $\frac{\partial u_i}{\partial b_{ij}}$ is the same for every j and proves the lemma. □

Theorem 1

Proof. To prove the theorem, we first design a new measure on the change rate of a particular sub-budget b_{ij} across two iterations. We use Δ_l to denote the ratio of $s_{ij}^{(l)}$ to $s_{ij}^{(l-1)}$, after lth iteration. Here, $s_{ij}^{(l)}$ and $s_{ij}^{(l+1)}$ are the probability after and before the iteration. That is,

$$\Delta_l = \max\left\{\frac{s_{ij}^{(l-1)}}{s_{ij}^{(l)}}, \frac{s_{ij}^{(l)}}{s_{ij}^{(l-1)}}\right\}.$$

Δ_l is always a positive real number no smaller than 1. When $\Delta_l = 1$, s_{ij}^{l} is equal to s_{ij}^{l-1}, implying the allocation of resource j w.r.t. job J_i stops changing. In the following, we prove that Δ_l tends to 1 in Algorithm 1. In particular, we study the relationship between Δ_l and Δ_{l+1}. There are two sub-cases possible to happen.

Case 1: If J_i is the one updating sub-budgets in lth iteration, we have

$$\frac{1}{\Delta_l} s_{ij}^{(l-1)} \le s_{ij}^{(l)} \le \Delta_l s_{ij}^{(l-1)}.$$

If $s_{ij}^{(l)} \ge s_{ij}^{(l-1)}$, $\frac{\partial u_i}{\partial s_{ij}^{(l)}} \le \frac{\partial u_i}{\partial s_{ij}^{(l-1)}}$, due to the non-increasing property of $\frac{\partial u_i}{\partial s_{ij}}$. On the other hand, combined with the fact that $1 - s_{ij}^{l} \le 1 - s_{ij}^{(l-1)}$, we have

$$\frac{\partial u_i}{\partial s_{ij}^{(l)}} s_{ij}^{(l)}\left(1 - s_{ij}^{(l)}\right) \le \Delta_l \frac{\partial u_i}{\partial s_{ij}^{(l-1)}} s_{ij}^{(l-1)}\left(1 - s_{ij}^{(l-1)}\right).$$

Since all other sub-budgets from other jobs remain the same, we have b'_{ij} is upper bounded by

$$\frac{b_i \Delta_l \frac{\partial u_i}{\partial s_{ij}^{(l-1)}} s_{ij}^{(l-1)}\left(1 - s_{ij}^{(l-1)}\right)}{\Delta_l \frac{\partial u_i}{\partial s_{ij}^{(l-1)}} s_{ij}^{(l-1)}\left(1 - s_{ij}^{(l-1)}\right) + \sum_{k\neq i}\left(\frac{\partial u_k}{\partial s_{kj}^{(l-1)}} s_{kj}^{(l-1)}\left(1 - s_{kj}^{(l-1)}\right)\right)},$$

and thus

$$b'_{ij} \le b_{ij}. \quad (A.2)$$

Then, by applying Eq. (1), we can derive the following ratio:

$$s_{ij}^{(l+1)} = \frac{\left(b'_{ij}\right)^{\alpha}}{\left(b'_{ij}\right)^{\alpha} + \sum_{k\neq i}\left(b_{kj}\right)^{\alpha}} < (\Delta_l)^{2\alpha} s_{ij}^{(l)}.$$

The ratio is thus upper bounded, i.e.

$$\Delta_{l+1} = \frac{s_{ij}^{(l+1)}}{s_{ij}^{(l)}} < (\Delta_l)^{2\alpha}.$$

Sub-Case 2: If J_k ($k \neq i$) is the one updating the sub-budgets, using similar strategy used in previous sub-case, we know $b'_{kj} \le \Delta_l b_{kj}$. Then, the probability s_{ij} is updated as

$$s_{ij}^{(l+1)} = \frac{\left(b_{ij}\right)^{\alpha}}{\left(b'_{kj}\right)^{\alpha} + \sum_{m\neq k}\left(b_{mj}\right)^{\alpha}} < (\Delta_l)^{\alpha} s_{ij}^{(l)}$$

If this is the case, we have $\Delta_{l+1} = \frac{s_{ij}^{(l+1)}}{s_{ij}^{(l)}} < (\Delta_l)^{\alpha}$.

Therefore, when $\alpha < 0.5$ and optimization iterations continue running for all jobs in round robin manner, the change ratio tends to be 1, i.e. $\lim_{l\mapsto\infty} \Delta_l = 1$.

Similarly, we can analyze the situation when s_{ij} decreases after the iteration. This will complete the proof that the algorithm must terminate after finite iterations. □

Lemma 3

Proof. By Lemma 2 and Lemma 1, Algorithm 1 always converges and the final results guarantees that the sub-budget is optimal on maximizing the utility of every job. Therefore, no matter what sub-budget the user submits, it is impossible to gain additional utility. □

Lemma 4

Proof. If the user submits another job J'_i with $b'_i < b_i$ and $u'_i = u_i$ such that J'_i achieves better utility than J_i, there exists an optimal sub-budget $\{b'_{i1}, b'_{i2}, \ldots, b'_{im}\}$ that $\sum_j b'_{ij} = b'_i$.

We can construct a new sub-budget that $\{b_{ij}\}$ with $b_{ij} = \frac{b_i}{b'_i} b'_{ij}$. Because $b'_i < b_i$ and $b_{ij} > b'_{ij}$ for each j. Due to the monotonicity property of the utility function, $\{b_{ij}\}$ must lead to better utility than $\{b'_{ij}\}$. Thus, the user can always gain additional utility by increasing the job's budget from b'_i to b_i, violating the original assumption. We thereby prove the lemma. □

Theorem 2

Proof. By Lemma 4, we have shown the user can improve the utility by bidding the resources using larger budget. Thus, there is no incentive for the user to bid with a smaller budget. In the rest of the proof, we show that telling the true utility function is also a dominating strategy for the user to maximize the utility.

Because of Lemma 4, it is safe to assume the user always tells the maximal budget b_i he can afford. Therefore, if the user submits a fake utility function u'_i instead of u_i for job J_i, Abacus will make a different sub-budget partitioning $\{b'_{i1}, \ldots, b'_{im}\}$ for job J_i, such that $\sum_j b'_{ij} = b_i$. The proof of Lemma 2 has already shown that the necessary condition for optimal sub-budget combination is keeping the same partial derivative with respect to every sub-budget. This observation implies that the only optimal solution for the sub-budget is $b'_{ij} = b_{ij}$ to ensure $\sum_j b'_{ij} = b_i$, where b_{ij} is the assignment result if user submits true utility function u_i. It is thus impossible for the user to gain additional utility by faking utility functions. □

Lemma 5

Proof. When $b'_i < b_i$, there are three possible cases, including (1) J_i remains in prolific pool; (2) J_i remains in non-prolific pool; and (3) J_i is moved from prolific pool to non-prolific pool. In the first case, the utility of the job decreases, according to Lemma 4, therefore contradicting the user's will on utility maximization. In the second case, the utility of job remains zero, as the job is not likely to be finished with the desired minimal utility. In the third case, the utility of the job definitely drops as well. It is undesirable for a user to submit $b'_i < b_i$.

Similarly, when $b'_i > b_i$, there are three possible cases, including (1) J_i remains in prolific pool; (2) J_i remains in non-prolific pool; and (3) J_i is moved from non-prolific pool to prolific pool. As discussed above, there is no incentive for the users when their jobs stay in the same pool, in the first two cases. Regarding the third case, the job will be finished with the utility, but the user needs to pay more than his budget. This is again a contradiction to the rationality assumption on the user. This completes the proof of the lemma. □

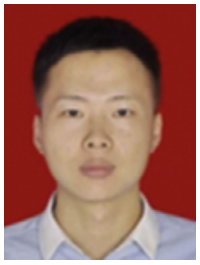
Lemma 6

Proof. When $m'_i > m_i$, there are three possible cases, including (1) J_i remains in prolific pool; (2) J_i remains in non-prolific pool; and (3) J_i is moved from prolific pool to non-prolific pool. In the first two cases, the utility of the job and the job payments both remain unchanged. In the third case, the utility of the job is negatively affected, as the utility of the job is no longer guaranteed. We can get similar results by adopting the same proving strategy on the case when $m'_i < m_i$. □

References

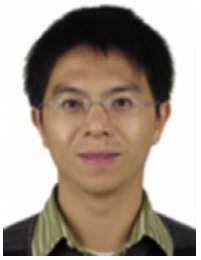
[1] Amazon elastic compute cloud (ec2), http://www.amazon.com/ec2.
[2] Apache hadoop, http://hadoop.apache.org
[3] Google app engine, http://appengine.google.com
[4] Mapreduce fair scheduler, http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html
[5] Microsoft windows azure, http://www.microsoft.com/windowsazure/
[6] Spark job scheduling, https://spark.apache.org/docs/1.4.0/job-scheduling.html
[7] G. Aggarwal, A. Goel, R. Motwani, Truthful auctions for pricing search keywords, in: Proceedings of ACM Conference on Electronic Commerce, 2006, pp. 1–7.
[8] G. Aggarwal, S. Muthukrishnan, D. Pál, M. Pál, General auction mechanism for search advertising, in: Proceedings of WWW, 2009, pp. 241–250.
[9] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, D. Tsafrir, Deconstructing amazon ec2 spot instance pricing, in: CloudCom, 2011, pp. 304–311.
[10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: a Berkeley view of cloud computing, Technical Report EECS-2009-28, UC Berkeley, 2009.
[11] C. Borgs, J.T. Chayes, N. Immorlica, K. Jain, O. Etesami, M. Mahdian, Dynamics of bid optimization in online advertisement auctions, in: Proceedings of WWW, 2007, pp. 531–540.
[12] J. -Y. Boudec, Rate adaptation, congestion control and fairness: a tutorial (2008). http://moodle.epfl.ch/file.php/523/CC_Tutorial/cc.pdf
[13] M. Brantner, D. Florescu, D.A. Graf, D. Kossmann, T. Kraska, Building a database on s3, in: Proceedings of Conference on SIGMOD, 2008, pp. 251–264.
[14] D. Buchfuhrer, S. Dughmi, H. Fu, R. Kleinberg, E. Mossel, C.H. Papadimitriou, M. Schapira, Y. Singer, C. Umans, Inapproximability for vcg-based combinatorial auctions, in: SODA, 2010, pp. 518–536.
[15] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, M. Wong, Tenzing a sql implementation on the mapreduce framework, PVLDB 4 (12) (2011) 1318–1327.
[16] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: OSDI, 2004, pp. 137–150.
[17] A. Demers, S. Keshav, S. Shenker, Analysis and simulation of a fair queueing algorithm, ACM SIGCOMM Computer Communication Review Vol. 19 (Issue 4) (1989) 1–12.
[18] B. Edelman, M. Ostrovsky, M. Schwarz, Internet advertising and generalized second price auction: selling billions of dollars worth of keywords, American Econ. Rev. 9 (1) (2007) 242–259.
[19] J. Feldman, S. Muthukrishnan, M. Pál, C. Stein, Budget optimization in search-based advertising auctions, in: Proceedings of ACM Conference on Electronic Commerce, 2007, pp. 40–49.
[20] A. Goel, K. Munagala, Hybrid keyword search auctions, in: WWW, 2009, pp. 221–230.
[21] H. Herodotou, F. Dong, S. Babu, No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, in: SOCC, 2011a.
[22] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin, S. Babu, Starfish: a self-tuning system for big data analytics, in: CIDR, 2011b, pp. 261–272.
[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, I. Stoica, Mesos: a platform for fine-grained resource sharing in the data center, in: NSDI, 2011.
[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: SOSP, 2009, pp. 261–276.
[25] F.P. Kelly, Charging and rate control for elastic traffic, Eur. Trans. Telecommun. 8 (1998) 33–37.
[26] A. Mehta, A. Saberi, U.V. Vazirani, V.V. Vazirani, Adwords and generalized on-line matching, in: FOCS, 2005, pp. 264–273.
[27] A.K. Parekh, R.G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single-node case, IEEE/ACM Trans. Networking 1 (3) (1993) 521–530.
[28] L. Popa, A. Krishnamurthy, S. Ratnasamy, I. Stoica, Faircloud: sharing the network in cloud computing, in: Hotnets, 2011.
[29] I. Raicu, Y. Zhao, I.T. Foster, A.S. Szalay, Data diffusion: dynamic resource provision and data-aware scheduling for data intensive applications, CoRR abs/0808.3535 (2008).
[30] T. Sandholm, K. Lai, MapReduce optimization using regulated dynamic prioritization, in: SIGMETRICS, 2009.
[31] M. Stillwell, D. Schanzenbach, F. Vivien, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, J. Paral. Distrib. Comput. 70 (9) (2010) 962–974.

[32] M. Stokely, J. Winget, E. Keyes, C. Grimes, B. Yolken, Using a market economy to provision compute resources across planet-wide clusters, in: IPDPS, 2009, pp. 1–8.
[33] J. Tan, X. Meng, L. Zhang, Delay tails in mapreduce scheduling, in: SIGMETRICS, 2012b, pp. 5–16.
[34] J. Tan, X. Meng, L. Zhang, Performance analysis of coupling scheduler for mapreduce/hadoop, in: INFOCOM, 2012a, pp. 2586–2590.
[35] B. Urgaonkar, P.J. Shenoy, T. Roscoe, Resource overbooking and application profiling in a shared internet hosting platform, ACM Trans. Internet Techn. 9 (1) (2009).
[36] H. Varian, Positon auction, Int. J. Indus. Org. 25 (2007) 1163–1178.
[37] A. Verma, L. Cherkasova, R.H. Campbell, Aria: automatic resource inference and allocation for mapreduce environments, in: ICAC, 2011, pp. 235–244.
[38] W. Vickrey, Counter-speculation, auctions and competitivesealed tenders, J. Finan. 16 (1) (1961) 8–37.
[39] L. Yang, J.M. Schopf, I.T. Foster, Conservative scheduling: using predicted variance to improve scheduling decisions in dynamic environments, in: SC, 2003, p. 31.
[40] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Job Scheduling for Multi-User MapReduce Clusters, Technical Report UCB-EECS-2009-55, EECS Department, University of California, Berkeley, 2009.
[41] Z. Zhang, R.T.B. Ma, J. Ding, Y. Yang, ABACUS: an Auction-Based Approach to Cloud Service Differentiation, in: IEEE IC2E, 2013. (best paper award).
[42] Z. Zhang, H. Shu, Z. Chong, H. Lu, Y. Yang, C-cube: elastic continuous clustering in clouds, in: ICDE, 2013.
[43] T.Z.J. Fu, J. Ding, R.T.B. Ma, M. Winslett, Y. Yang, Z. Zhang, DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams, in: IEEE ICDCS, 2015.



Jianbing Ding is currently a Ph.D. student in the School of Data and Computer Science, Sun Yat-Sen University. He received his B.E. degree from Nanjing University of Post and Telecommunications in 2008, and master degree from the University of Science and Technology of China in 2011. His research interests are mainly in cloud computing and large scale information retrieval systems.

Zhenjie Zhang is currently working as a Research Scientist at the Advanced Digital Sciences Center. He received his Ph.D. from the School of Computing, National University of Singapore, in 2010. His research interests cover a variety of different topics, including elastic streaming processing, non-metric indexing and data privacy. He has served as a Program Committee member for WWW, VLDB, KDD, SIGMOD, ICDE.



Richard T. B. Ma received the B.Sc. (first-class honors) degree in Computer Science in July 2002 and the M.Phil. degree in Computer Science and Engineering in July 2004, both from the Chinese University of Hong Kong, and the Ph.D. degree in Electrical Engineering in May 2010 from Columbia University. During his Ph.D. study, he worked as a research intern at IBM T.J. Watson Research Center in New York and Telefonica Research in Barcelona. He is currently a Research Scientist in the Advanced Digital Sciences Center, University of Illinois at Urbana-Champaign and an Assistant Professor in School of Computing at National University of Singapore. His current research interests include distributed systems, network economics, game theory, and stochastic processes.

Yin "David" Yang is currently an Assistant Professor in the College of Science and Engineering, Hamad Bin Khalifa University. His main research interests include cloud computing, database security and privacy, and query optimization. He has published extensively in top venues on differentially private data publication and analysis, and on query authentication in outsourced databases. He is now working actively on cloud-based big data analytics, with a focus on fast streaming data.