ABACUS: An Auction-Based Approach to Cloud Service Differentiation

Zhenjie Zhang [1], Richard T. B. Ma [1,2], Jianbing Ding [3], Yin Yang [1]

[1]*Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore*
{zhenjie, yin.yang}@adsc.com.sg

[2]*School of Computing, National University of Singapore, Singapore*
tbma@comp.nus.edu.sg

[3]*School of Software, Sun Yat-Sen University, China*
dingsword@gmail.com

***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. An extensive set of experiments, running on Hadoop, demonstrate the high performance and other desirable properties of Abacus.**

I. 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 [11], [13], [14].

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 *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 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.

In the following, Section II reviews related work. Section III provides problem definition and assumptions. Section IV details the auction mechanism. Section V discusses automatic optimization for users not knowing their own utility functions. Section VI contains an extensive experimental evaluation. Finally, Section VII concludes the paper.

II. Related Work

A. General Resource Allocation

Resource allocation [24], [10] and scheduling [28], [16] 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) [16] and Generalized Process Sharing (GPS) [28], achieve a *max-min fair* [10] 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], [4] and the research community [8], little work addresses the issue of providing service differentiations to users.

Hindman et al. [22] 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. [29] 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. [7], [31], but without theoretical analysis on the effectiveness in game environment.

B. Map-Reduce Scheduling

MapReduce [15] 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. [35] 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 [35]. Issard et al. [23] independently propose a scheduling strategy similar to [35]. 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].

Lee et al. [25], [26] aim at improving system efficiency by considering low-level system information, such as the topology of the computer cluster. Herodotou etl al. [20], [21] 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.

C. Auction Mechanism

Recently, auctions have been successfully applied to online keyword advertising. Specifically, each advertiser submits a bidding prices 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 [17], [33]. This has propelled the research from both economics and computer science communities [5], [6], [19], [27], 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) [34] and Generalized Second Price (GSP) [33]. Several studies also consider budget constraints for the advertisers, e.g., [9], [12], [18], [37]. 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.

III. 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.

Abacus allows each user to submit multiple jobs. 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. Assume that $s_i = (s_{i1}, s_{i2}, \ldots, s_{im})$ is the vector of resources assigned to job J_i, in which each s_{ij} is an assignment probability 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. In Amazon EC2, for example, 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, with a lower level of service quality. Therefore, the utility function on EC2 can be written as a linear utility function, with w_j as the usefulness of a virtual machine of type j. In particular, the job gains positive utility, even when there is no resource assigned on certain resource, 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. The results in this paper is valid for both linear and logscale utility functions, and, in general, any utility function satisfying the condition of *sum of non-decreasing concave functions*.

Given the utility function, the user submits his job J_i to Abacus. 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 proportional to 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 to finish.

There are two important components in Abacus, *Auctioneer* and *Scheduler*. Figure 1 presents the relationship be-

tween these components, in the context of the MapReduce 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 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.

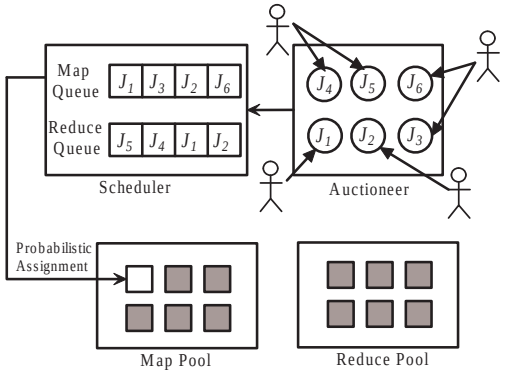


Fig. 1. Abacus system architecture on Hadoop

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 [22], 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. We design some probabilistic assignment protocol for the auctioneer. In the following section, we will present the auction mechanism and prove the desirable properties of our mechanism.

IV. Auction-Based Protocol

This section describes the auction mechanism in Abacus. Section IV-A introduces the basic procedure of the auction mechanism. Section IV-B presents the algorithm on the assignment probability computation. Section IV-C proves the desirable economical properties of the mechanism.

A. Basic Protocol

Recall from Section III 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 Figure 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 $u_3(x, y) = 2x + 4y$.

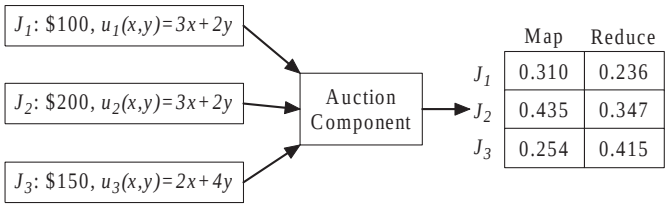


Fig. 2. Example of auction with three jobs on Hadoop

B. 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. $s_{ij} = 1/n$ 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:* [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}.

[1]Due to the space limit, we omit all the proofs of our theoretical results. However, they are available in our technical report [36].

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 Figure 3, Abacus partitions the budgets based on the jobs in Figure 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.

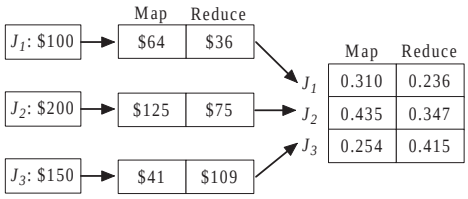


Fig. 3. Example of sub-budget partitioning

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 [17]. The details of the analysis will be covered in the rest of this section.

The auction mechanism employed in Abacus aims to find such a matrix $S = M(\{J_1, \ldots, J_n\})$. 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 $\left\{b_{i1}^{(t)}, \ldots, b_{im}^{(t)}\right\}$ 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** $\exists i, j, \quad \|b_{ij}^{(t)} - b_{ij}^{(t-1)}\|_2 \geq \epsilon$ **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 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.

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 $\{b'_{i1}, \ldots, b'_{im}\}$

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 Figure 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. Deeper investigation on α is left for future work.

C. 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

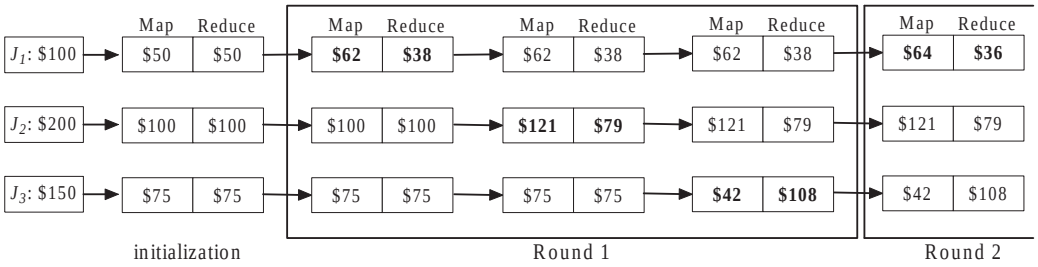


Fig. 4. Auctioneer automatically optimizes the sub-budgets of all jobs in order, until no job can further improve any more.

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 [17].

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 Figure 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. In the future, we are interested in analyzing the behavior of multiple jobs from an individual user and its impact on the system performance.

V. Utility Function Estimation

In practice, an ordinary user of the cloud system may not know the exact utility function of his jobs and services. It is thus not practical for most cloud systems to force the users to submit budgets and utility functions before running their jobs. To alleviate the problem, we derive a novel approach in this section to automatically estimate the utility function when similar jobs from a single are repeatedly run. This enables the users to participate in the auction with their budget knowledge only. Our estimation is purely based on analysis over the utilities/performance of the jobs/service 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 III, i.e. *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* system allows the user to submit the budget only but without 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 round 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.

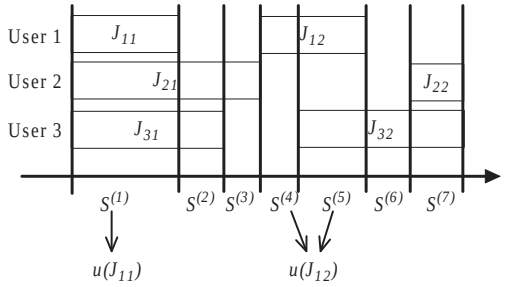
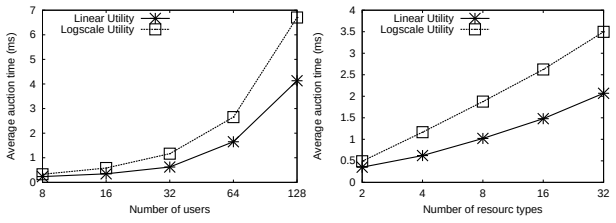


Fig. 5. An example of running jobs with three users, in which u_1's utility function is possibly wrong.

In Figure 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



(a) varying # of users (b) varying # of resource types

Fig. 7. Experiments on the computation time of auction algorithm

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 at 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. $\overline{s}_{il} = (\overline{s}_{il1}, \ldots, \overline{s}_{ilm})$, such that $\overline{s}_{ilj}$ 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 $\{\overline{s}_{i1}, \ldots, \overline{s}_{ik}\}$. Given such statistic information, we run a regression to find out the optimal weights to minimize the following objective function.

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

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.

VI. Experiments

In this section, we conduct empirical studies on the performance and properties of *Abacus* in a real cloud system.

A. Experimental Setup on Real 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 2.4GHz CPU, 48GB RAM, two 146GB SAS hard disks and another two 500GB SAS hard disks. Each slave node has a single-core 2.4GHz CPU, 8GB RAM and a 500GB SATA hard disk. All the nodes used in the cluster are running CentOS 5.5 and Hadoop 0.20.

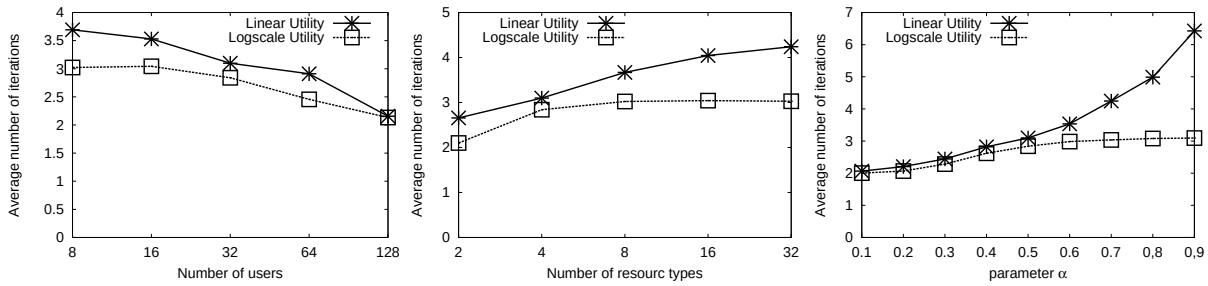
Abacus on Hadoop: To test the usefulness of *Abacus* on practical cloud systems, we added a new auction component into Hadoop 0.20, by redesigning the scheduler in Hadoop based on *Fair Scheduler* [35]. Basically, our meta scheduler monitors the profiles as well as the submitted budgets of the active 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 [35] to ensure effective resource allocation and scheduling. The source codes of the implementation and installation instructions on Hadoop 0.20 system are available at https://sites.google.com/site/zhangzhenjie/abacus.zip.

B. 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. All of the numbers reported in this subsection are averages over 1,000 random runs. The default setting of the experiments are: 32 system users, 4 types of resources, and balancing parameter $\alpha = 0.5$. The budgets of the users follow an independent and identical uniform distribution on 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 $[0.5, 2]$.

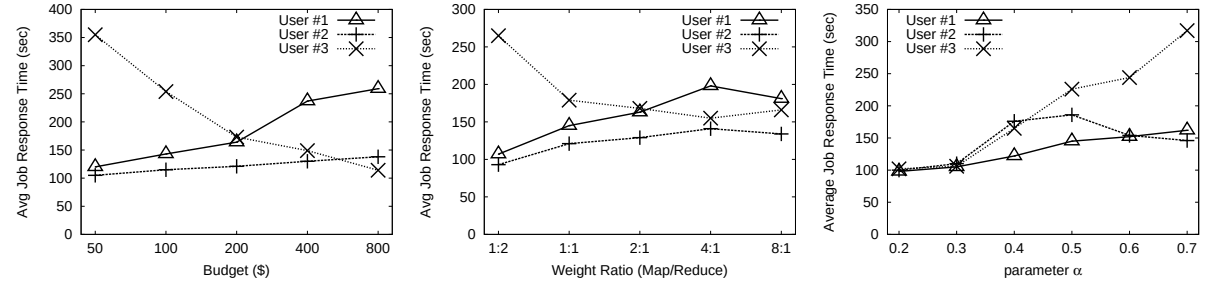
Figure 6 shows the average number of iterations of the auction algorithm under different settings. The auction algorithm generally converges very quickly, within 5 iterations in most settings. When there are more users in the auction, as is shown in Figure 6(a), the convergence rate is accelerated. This 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 i rounds of updates on the resource allocation. When increasing the number of resource types (Figure 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 Figure 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

[2]http://www.comp.nus.edu.sg/~epic



(a) Number 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



(a) Budget of user #3 vs. response time (b) Utility Function of user #3 vs. response time (c) α vs. response time

Fig. 8. Experiments on the Incentive Compatibility property

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.

Figure 7 evaluates the total computation time of the auction. When increasing the number of active users (Figure 7(a)) or the number of resource types (Figure 7(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 milliseconds by average, which is negligible. This implies that the proposed auction algorithm can handle a large number of job arrivals or departures.

C. Incentive Compatibility

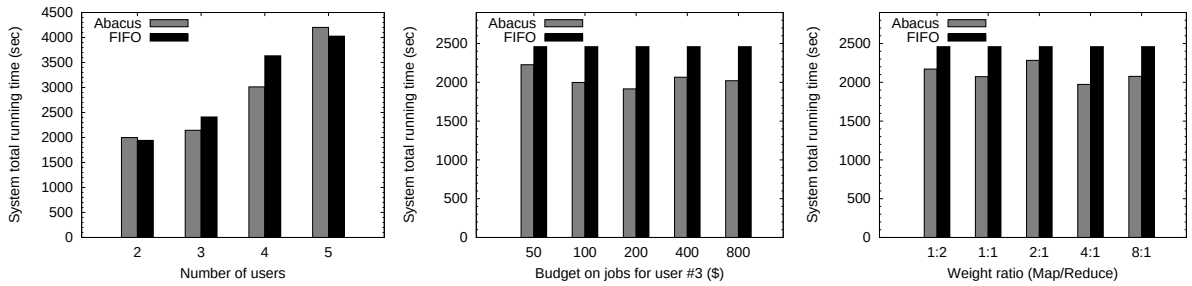
Next we verify the incentive compatibility property of *Abacus* on real systems. To do this, we simulated three different users that submit different types of MapReduce jobs. When generating the workload, we assumed that job arrivals of the three users follow three independent Poisson processes with 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 I, 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 minutes from the beginning of the experiments. The system stops after finishing all submitted jobs. We use the inverse of job response time as the measure for job utility. The actually 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 [32], our experimental results imply that it is sufficient to model the job performance when most of the jobs in the system are *short* jobs.

TABLE I
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 most important implication of incentive compatibility is that the users maximize their job efficiency by submitting the maximum affordable budgets and true utility 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 Figure 8(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 deteriorates, since their bids are weakened when another participant spends more budget.

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



(a) # of users vs. system efficiency (b) Budget of user #3 vs. system efficiency (c) Util. func. of user #3 vs. system efficiency

Fig. 9. Experiments on system efficiency

(Figure 8(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 I, the jobs from user #3 spend about 60 seconds on maps and 10 seconds on reduces. Thus, the true ratio of map to reduce weight is about 6:1. In Figure 8(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 are running slower when user #3 is reporting his true utility.

In Figure 8(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.

D. 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 negligible overhead on the overall performance of the system. Figure 9 shows that Abacus has competitive performance compared to FIFO, when varying the number of users joining the resource auction. There are at most 5 users in our experiments, due to the limited computation resources in our cluster. 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 the throughput. The only exception happens when there are 5 users, in which the completion time of Abacus with all jobs is slightly worse (at most 5% more) compared to FIFO.

In Figure 9(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. Figure 9(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.

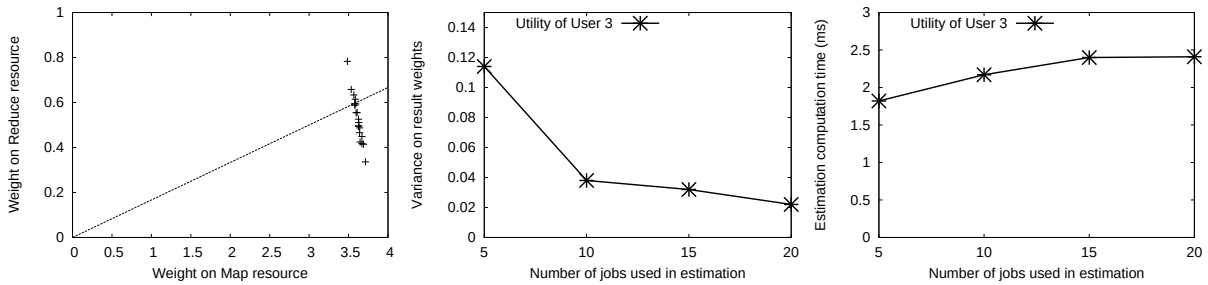
E. Utility Function Estimation

Finally, we evaluate the effectiveness of utility function estimation technique described in Section V. 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 VI-C. 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 Figure 10(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). Figure 10(b) reports the variance of the result when varying the number of jobs. Clearly, the variance plunges as the number of jobs increases. Figure 10(c) shows that all utility function estimations finish within 3 milliseconds, which is negligible.

VII. 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



(a) Distribution of the result utility function (b) Variance of the result utility function (c) Computation time of the estimation

Fig. 10. Experiments on utility function estimation

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.

ACKNOWLEDGMENTS

Zhang, Ma and Yang are partly supported by the research grant for the Human Sixth Sense Programme at the Advanced Digital Sciences Center from Singapores Agency for Science, Technology and Research (A*STAR). Ma is also supported by the Ministry of Education of Singapore AcRF grant R-252-000-448-133.

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] Microsoft windows azure. *http://www.microsoft.com/windowsazure/*.
[5] G. Aggarwal, A. Goel, and R. Motwani. Truthful auctions for pricing search keywords. In *ACM Conference on Electronic Commerce*, pages 1–7, 2006.
[6] G. Aggarwal, S. Muthukrishnan, D. Pál, and M. Pál. General auction mechanism for search advertising. In *WWW*, pages 241–250, 2009.
[7] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. In *CloudCom*, pages 304–311, 2011.
[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. *Technical Report EECS-2009-28, UC Berkeley*, 2009.
[9] C. Borgs, J. T. Chayes, N. Immorlica, K. Jain, O. Etesami, and M. Mahdian. Dynamics of bid optimization in online advertisement auctions. In *WWW*, pages 531–540, 2007.
[10] J.-Y. Boudec. Rate adaptation, congestion control and fairness: A tutorial. 2008.
[11] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD Conference*, pages 251–264, 2008.
[12] D. Buchfuhrer, S. Dughmi, H. Fu, R. Kleinberg, E. Mossel, C. H. Papadimitriou, M. Schapira, Y. Singer, and C. Umans. Inapproximability for vcg-based combinatorial auctions. In *SODA*, pages 518–536, 2010.
[13] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es^2: A cloud data storage system for supporting both oltp and olap. In *ICDE*, pages 291–302, 2011.
[14] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.
[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[16] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review, Vol. 19, Issue 4, 1989*.
[17] B. Edelman, M. Ostrovsky, and M. Schwarz. Internet advertising and generalized second price auction: Selling billions of dollars worth of keywords. *American Economics Review*, 9(1):242–259, 2007.
[18] J. Feldman, S. Muthukrishnan, M. Pál, and C. Stein. Budget optimization in search-based advertising auctions. In *ACM Conference on Electronic Commerce*, pages 40–49, 2007.
[19] A. Goel and K. Munagala. Hybrid keyword search auctions. In *WWW*, pages 221–230, 2009.
[20] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SOCC*, 2011.
[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.
[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, pages 261–276, 2009.
[24] F. P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1998.
[25] G. Lee, B.-G. Chun, and R. H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *HotCloud*, 2011.
[26] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. Topology-aware resource allocation for data-intensive workloads. In *ApSys*, pages 1–6, 2010.
[27] A. Mehta, A. Saberi, U. V. Vazirani, and V. V. Vazirani. Adwords and generalized on-line matching. In *FOCS*, pages 264–273, 2005.
[28] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):521–530, June 1993.
[29] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Hotnets*, 2011.
[30] T. Sandholm and K. Lai. MapReduce optimization using regulated dynamic prioritization. In *SIGMETRICS*, 2009.
[31] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. Using a market economy to provision compute resources across planet-wide clusters. In *IPDPS*, pages 1–8, 2009.
[32] J. Tan, X. Meng, and L. Zhang. Performance analysis of coupling scheduler for mapreduce/hadoop. In *INFOCOM*, pages 2586–2590, 2012.
[33] H. Varian. Positon auction. *International Journal of Industrial Organization*, 25:1163–1178, 2007.
[34] W. Vickrey. Counter-speculation, auctions and competitivesealed tenders. *Journal of Finance*, 16(1):8–37, 1961.
[35] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. Technical Report UCB-EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.
[36] Z. Zhang, R. T. B. Ma, J. Ding, X. Xiao, and Y. Yang. Abacus: An auction-based approach to cloud service differentiation. Technical report, National University of Singapore, http://www.comp.nus.edu.sg/~tbma/abacus-TR.pdf, Sept. 2012.
[37] Y. Zhou, D. Chakrabarty, and R. M. Lukose. Budget constrained bidding in keyword auctions and online knapsack problems. In *WINE*, pages 566–576, 2008.