Cocoa: Dynamic Container-based Group Buying Strategies for Cloud Computing

Xiaomeng Yi and Fangming Liu*, Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology
Di Niu, University of Alberta
Hai Jin, Key Laboratory of Services Computing Technology and System, Ministry of Education, School of Computer Science and Technology, Huazhong University of Science and Technology
John C.S. Lui, The Chinese University of Hong Kong

Although the Infrastructure-as-a-Service cloud offers diverse instance types to users, a significant portion of cloud users, especially those with small and short demands, cannot find an instance type that exactly fits their needs or fully utilize purchased instance-hours. In the meantime, cloud service providers are also faced with the challenge to consolidate small short jobs, which exhibit strong dynamics, to effectively improve resource utilization. To handle such inefficiencies and improve cloud resource utilization, we propose *Cocoa*, a novel group buying mechanism that organizes jobs with complementary resource demands into groups and allocates them to group buying deals predefined by cloud providers. Each group buying deal offers a resource pool for all the jobs in the deal, which can be implemented as either a virtual machine or a physical server. By running each user job on a virtualized container, our mechanism allows flexible resource sharing among different users in the same group buying deal, while improving resource utilization for cloud providers. To organize jobs with varied resource demands and durations into groups, we model the initial static group organization as a variable-sized vector bin packing problem, and the subsequent dynamic group organization problem as an online multidimensional knapsack problem. Through extensive simulations driven by a large amount of real usage traces from a Google cluster, we evaluate the potential cost reduction achieved by *Cocoa*. We show that through the effective combination and interaction of the proposed static and dynamic group organization strategies, *Cocoa* greatly outperforms existing cloud workload consolidation mechanism, substantiating the feasibility of group buying in cloud computing.

CCS Concepts: •**Computer systems organization** → **Cloud computing;** •**Theory of computation** → **Packing and covering problems; Online algorithms;**

Additional Key Words and Phrases: Group buying, container, cost saving

ACM Reference Format:
Xiaomeng Yi, Fangming Liu, Di Niu, Hai Jin, John C.S. Lui, 2016. Cocoa: Dynamic Container-based Group Buying Strategies for Cloud Computing. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 9, 4, Article 39 (March 2010), 30 pages.
DOI: 0000001.0000001

1. INTRODUCTION

The Infrastructure-as-a-Service (IaaS) cloud delivers bundled resources such as CPU, memory, and storage in the form of virtual instances, and promises to save cost for users in a pay-as-you-go model. By consolidating virtual instances to occupy fewer

The Corresponding Author is Fangming Liu (fmliu@hust.edu.cn). This work was supported in part by the National Natural Science Foundation of China under Grant 61520106005, in part by the National Key Research and Development Program under grant 2016YFB1000501, and in part by the National 973 Basic Research Program under Grant 2014CB347800.

DOI: 0000001.0000001

physical servers, cloud service providers can also enhance resource utilization. Unfortunately, these promises of IaaS are not fulfilled when dealing with small jobs with low resource demands and short durations. Most small jobs have resource demands that seldom exactly fit into the offered instance types, and may thus have to pay for resources they do not use. Moreover, small jobs may have short durations of only several minutes [Wang et al. 2013], and thus cannot fully utilize the entire instance period (*e.g.*, an hour) they pay for. Although CloudSigma [2014] allows users to customize their instances and Google Compute Engine charges users in a per-minute manner, most cloud providers offer fixed instance types and charge users on an hourly basis.

For cloud service providers, small jobs pose unique challenges to cloud workload consolidation, which is traditionally done by packing virtual machines (VMs) of complementary demands onto the same physical server [Meng et al. 2010]. However, small jobs exhibit strong dynamics, even in their aggregated demand. The provider does not always have jobs with complementary resources to pack onto a physical server. For example, during those times when most user jobs are CPU-bound, VM consolidation to improve memory utilization cannot be effectively performed, since it would bring a high risk of service violation in terms of CPU. Moreover, the short job duration and churning further make it difficult to consolidate VMs.

We propose *Cocoa* (COmputing in COntAiners), a lightweight container-based "group buying" mechanism, to attract and consolidate small users with complementary resource demands in cloud computing. Group buying is a marketing strategy that emerged in the last decade and has become popular in e-commerce. In group buying, consumers enjoy a discounted group price if they form a group to purchase a deal. Similarly, using price discount incentives, a cloud service provider can offer various group buying deals to attract users with complementary demands to form groups. If such users can indeed coordinate with each other and collectively submit requests, consolidation is naturally performed with a better utilization of physical resources, and thus the price discount offered in the first place can be warranted.

Thanks to the emerging *container-based virtualization* [Strauss 2013], cloud group buying can be realized on existing IaaS platforms by the cloud provider (or a third-party service broker) in a lightweight fashion. It is traditionally believed that virtualization based on VMs is the only way to provide isolation for applications running on a server. However, this assumption is quickly becoming outdated, as *containers* can now be used as an alternative to VM-based virtualization to run multiple isolated applications on a single operating system, with shared binary and library resources. Unlike VMs, containers are much more efficient, as they do not require a full operating system image for each job, and have such attractive features as lower overhead, shorter boot time, higher consolidation ratio, and more flexible manageability. Therefore, in our group buying mechanism, we load each small job onto a container, and pack such containers into the predefined "group buying deals", implemented as VMs or directly as Physical Machines in the existing IaaS infrastructure.

Unlike traditional VM consolidation for stable large jobs [Xiao et al. 2013], we face the unique challenge to group small and short workloads that may come and go dynamically. Live migration [Xu et al. 2014b], which is widely used in VM consolidation to cope with workload dynamics, would be too costly to group small user jobs, considering their short durations and strong dynamics. Therefore, we combine both static and dynamic grouping strategies in *Cocoa*. At the core of *Cocoa*, there is a job scheduler that not only deals with the variety of user demands with a static batch-based grouping strategy, but also handles job dynamics by packing user jobs into groups on the fly. We formulate the initial static batch-based grouping as a *variable-sized vector bin packing* problem, which turns out to be an NP-hard integer program that can be approximated in polynomial time. The performance of the approximation scheme proves

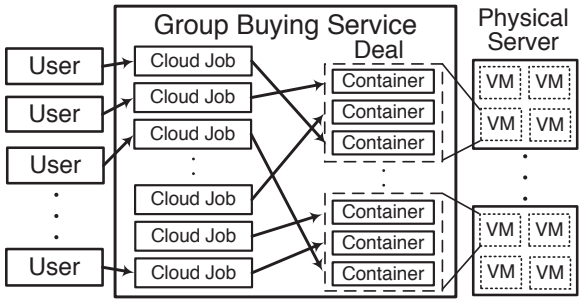

Fig. 1. The framework of proposed container-based group buying service.

to be close to optimal in our simulation. More importantly, with job dynamics, there will be many resource "holes" left in the group buying deals due to job departures. To pack newly arrived jobs into these resource "holes" in an online fashion, we formulate the hole-filling process as an online knapsack problem to enhance utilization, and describe a practical online scheme to handle job dynamics.

As a highlight of this paper, we conduct extensive trace-driven evaluations of the proposed mechanisms based on the workload traces collected from more than 12,000 Google servers in a 29 day period. We carefully evaluate the cost saving percentage, resource utilization as well as service delay of *Cocoa* under different parameter settings. The simulation results demonstrate that *Cocoa* can bring a significant amount of cost savings to both cloud providers and users at the cost of only a moderate service waiting time.

2. DESIGN PHILOSOPHIES

In this section, we provide an overview of our group buying mechanism. In reality, restaurants may use *Groupon*, *LivingSocial* or *Tuangou* (a web-based group buying service based in China) to attract group purchases to increase the total sale, while exploiting the reduced cost of serving a same predefined three-course meal in large quantity. Using a similar idea, cloud service providers can offer group buying deals, each aimed at accommodating a bunch of small users together, such that the sum of the user resource requirements in all resource dimensions will not exceed the capacity specified in the corresponding group buying deal.

Our cloud group buying framework is illustrated in Fig. 1, where each user job runs on a container and multiple containers can be placed onto a group buying deal, which may be implemented as a VM or even a physical machine. The job organizations are done through a smart interaction between static grouping and dynamic grouping strategies.

Economic Incentives. Group buying can save costs for both users and cloud providers due to several reasons. *First*, given limited instance types, the resource demand of small jobs seldom exactly match any given instance. In most cases, only part of resources in an instance paid for is used. For example, a memory-intensive task on a small instance of Amazon EC2 may use only a small portion of the available CPU resources. When jobs with complementary demands are grouped together, resource wastage will be reduced, thus saving cost to both the cloud provider and users. *Second*, many small jobs only run for partial hours on an instance or have intermittent resource usage. With the currently most popular hourly charging model, such users are actually charged for the whole instance-hour. To handle job dynamics, our group buying mechanism not only packs user demands along different resource dimensions, but also along the timeline. After a job finishes and departs from the group, its container will be eliminated and the corresponding resources are released to accommodate new

jobs. When a new job is allocated to the group, a container can be launched in a few seconds to quickly recycle idle resources.

Despite its potential benefits to both cloud providers and users, such benefits cannot be realized without smart grouping strategies. Similar to existing group buying services like Groupon and LivingSocial, we propose a new type of business model to be operated by cloud providers. In this model, each small user interested in the group buying service needs to submit his/her (rough) resource request for CPU cores (or "compute cycles" in the case of RackSpace), memory, and other resources. Note that such resource request can take any value, not necessarily the ones in predefined VM types. The cloud provider predefines several group buying deals, each with certain (relatively large) resource provisioning. At the core of *Cocoa* is a scheduler, operated by the cloud provider, which packs different jobs into as few group buying deals as possible so that the total service cost is minimized.

Application Scenario. In *Cocoa*, user submitted jobs may need to wait for a period of time before it is allocated to a proper deal and be executed. Therefore, *Cocoa* is more suitable to serve non-interactive jobs. One application example would be job batches for routine system maintenance such as periodical log analysis and data backup. In addition, recent literature [Elmeleegy 2013] indicates that in modern Hadoop-like big data systems [Yi et al. 2014], a large portion of the workloads are small and short, which could also be served in *Cocoa*. This is mainly because those systems are usually used as data warehouses to serve ad-hoc, short queries generated by higher level query languages, which can typically be completed within a couple of minutes [Chen et al. 2012]. The provider could provide an interface for job submission, which allows users to specify the software stack and the amount of computing resources required by the job, as well as a shell script to run the job. For a submitted job, the cloud provider allocates it to a proper group buying deal with either static or dynamic grouping strategy, and launchs a container in the deal according to the user specified requirements. Once the container is launched, the provider can execute the user submitted shell script to run the job. In addition to the compute service, modern cloud service providers also offer cloud storage services (*e.g.,* Amazon EFS and Microsoft Azure Storage), which can be accessed from user launched container. Therefore, cloud users could upload job required files (*e.g.,* executable files and input data) to the cloud storage service upon job submission. When the job is finished, the results can either be stored in the cloud storage service or be transferred to a local on-premise server of the user.

The benefits of cloud group buying do not come without concerns. In the group buying market, users need to wait for a period of time to join in a group and get served, especially in the batch-based grouping. We will show in Sec. 4 and Sec. 6 that by using the dynamic grouping strategies, the service delay can be reduced to a moderate extent. In addition, in our group buying mechanism, cloud users are required to submit their resource requests before job execution, which may not be an accurate estimate. However, demand estimation is also required in individual buying. As will be mentioned later, container-based multiplexing in the same group can help alleviate this problem.

In general, cloud group buying gives users an opportunity to execute small and short jobs at a lowered price, at the cost of a compromise in service delay. However, our container-based group buying should be regarded as a complement to the current "individual buying" mechanism rather than its competitor. For users with time-critical tasks, or those who need to deal with workload spikes by dynamically scaling their instances, traditional IaaS may still be a better choice.

Containerization, Isolation and Multiplexing. We run each job inside a group buying deal on a *container* to achieve isolation. Containers can now be used as an alternative to VM-based virtualization to run multiple isolated systems on a single

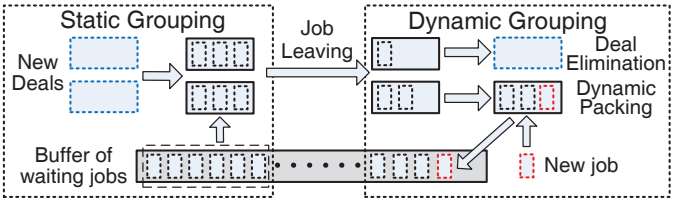

Fig. 2. The static group buying strategy packs a batch of jobs in the waiting buffer into new opened deals. The dynamic strategy tries to pack each new job into resource "holes" in existing group buying deals caused by job departure. If there is no deal that a job can fit into, it will be added to the waiting buffer. Empty deals with no job running will be eliminated.

host. Unlike VMs, containers do not require a full OS image for each job, multiple containers can share a single operating system, with little overhead associated with job launching or termination. Therefore, the lightweight containers are more suitable to handle job dynamics as compared to VMs. As containers within a single OS are much more efficient, they may underpin the future of the cloud infrastructure industry in place of VM architecture [Strauss 2013]. In fact, cloud providers such as Google Cloud Platform, RackSpace and dotCloud [Petazzoni 2013] have already adopted containers to achieve isolation among hosted cloud applications. It is also reported that Google is now running all its applications in containers [Morgan 2014]. In addition, several leading IaaS vendors like AWS EC2, Google Compute Engine and RackSpace Cloud support containers to run on their virtual machine instances [Docker 2014]. Therefore, our group buying mechanism can be deployed directly upon existing IaaS platforms, by implementing group buying deals as predefined VM instances, on which multiple containers can be launched to run jobs.

The physical resources allocated to each container are configurable. For each container, we set its resource upper limit to the user's requested resources. However, the actual resource usage of each job may not be exactly the same as its original estimation. The actual resource demand of a job may exceed its container's upper limit, while some other job may not use up all the resources allocated to its container. Fortunately, container schedulers have some features that allow different containers on the same host to share resources more flexibly. For example, OpenVz [2014] can monitor container resource usage and resize them accordingly with a low overhead. Some LXC-based containers [Soltesz et al. 2007] allow unused resource in one container to be used by other containers. And Parallels Virtuozzo containers [virtuozzo 2014] are able to use more resources than they are initially configured when there are idle resources in the system. Since not all jobs will have resource peaks at the same time, jobs in the same group buying deal can multiplex their bursty demand to further save cost.

Static vs. Dynamic Group Buying Strategies. As illustrated in Fig. 2, to pack user jobs into group buying deals, *Cocoa* combines the usage of two grouping strategies, *i.e.,* the static grouping strategy and the dynamic grouping strategy. Initially, *Cocoa* uses a static strategy to pack a batch of jobs into group buying deals. After that, there will be both incoming jobs due to new submission and leaving jobs due to job completion. When a job departs from a deal, its container will be eliminated (*i.e.,* delete container image and release its allocated sources), thus leaving the corresponding resources idle. To avoid wastage, *Cocoa* packs newly arrived jobs into existing group buying deals to fill the resource "holes". Upon the arrival of a new job, *Cocoa* uses dynamic grouping strategy to decide whether to pack it into existing deals and also which deal to pack it into. If no running deal can accommodate the new job, then it will be put into a waiting buffer. When there are sufficient number of waiting jobs in the buffer, the provider will resort to static grouping strategy to pack all of the waiting jobs into

new group buying deals. In summary, the static strategy is used to pack a batch of waiting jobs into new created deals, and the dynamic grouping strategy packs newly arrived jobs into existing group buying deals.

Note that *Cocoa* immediately performs dynamic grouping upon job arrival. Thus, for dynamically grouped jobs, they would only experience a computation delay when *Cocoa* calculates the decision about which running deal to pack the job in. However, in addition to the computation delay, jobs that are statically grouped will experience an extra service queuing delay, since it has to wait for a certain amount of time in the waiting buffer before being packed. Since all the jobs in *Cocoa* are delay tolerant, in static grouping, we focus on minimizing the total deal cost to serve jobs while neglecting the service latency. In *Cocoa*, to avoid high latency, we also set an upper bound for the waiting time and immediately perform the static grouping when the waiting time of any job reaches the upper bound. In Sec. 6.6, we conduct trace-driven simulation to further demonstrate that the dynamic grouping strategy effectively avoids the queuing delay for most of jobs in *Cocoa*.

It is also worth noting that the static and dynamic grouping strategies share the same objective, which is to minimize the cost of serving all the jobs. In static grouping, we pack a batch of jobs into as few deals as possible to minimize the cost of launching deals. Therefore, it is modeled as a bin-packing problem as elaborated in Sec. 3. In dynamic grouping, we pack newly arrived jobs into existing deals. By setting the objective as maximizing the utilization of idle resources in running deals, we not only minimize wastage of idle resources in running deals but also maximize the sum of resource demand from jobs that are packed into existing deals. It also minimizes the sum of resource demands from all the jobs that are not packed into existing deals, since the total resource demand of newly arrived jobs (both packed and not packed jobs) is fixed. Note that *Cocoa* creates new deals to serve jobs that are not packed in dynamic grouping. By minimizing the total resource demands of not packed jobs, the dynamic grouping strategy minimizes the need to open up new deals, thus saving the service cost. For each new job, the dynamic grouping strategy first decides whether to pack the job into existing deals or let it wait for being statically grouped into new deals. Then for the jobs that should be packed into existing deals, it also needs to decide which deal to pack in. As will be elaborated in Sec. 4, the dynamic group problem is more suitable to be modeled as an online knapsack problem.

3. STATIC GROUPING FOR BATCHES OF JOBS

Cocoa combines the usage of static and dynamic strategies to pack user jobs into predefined group buying deals. The static grouping strategy is performed to pack a batch of waiting jobs at the beginning of the group buying service, or whenever there is a sufficient number of jobs waiting to join buying groups. In Sec. 3.1, we formulate the static group buying problem as a variable-sized vector bin packing problem. We adapt the literature of a branch-and-price framework to our problem and introduce an optimal solution with relatively high computation complexity in Sec. 3.2. Based on the framework, in Sec. 3.3, we present a polynomial-time approximation algorithm for the static grouping problem.

3.1. Problem Formulation

In our proposed group buying mechanism, a cloud provider can predefine multiple types of group buying deals, each with specification on the amount of resources in all dimensions, as well as the price of the deal. We assume the capacities of cloud providers are sufficiently large such that deals of each type can always be provided upon request.

We consider a batch of m waiting jobs. There are n types of group buying deals in the market. The computing resources (*e.g.*, CPU, memory, I/O, *etc.*) considered have

a dimension of d. We use a vector $C_j = (c_{1j}, c_{2j}, \ldots, c_{dj})$ to denote the computing resources provided in a group buying deal of type-j, with $j \in \{1, \ldots, n\}$, where c_{kj} is the amount of dimension-k computing resource offered in a type-j deal. In practice, cloud service providers offer multiple types of instances, some of the instance types provide relatively large amounts of computation resources, which can be used as group buying deals. For instance, a m4.10xlarge instance in Amazon EC2 contains 40 vCPUs and 160 GB memory. By running it as a group buying deal, we can launch multiple containers to run cloud jobs concurrently. Assume that the cloud provider implements each type-j group buying deal by launching a VM instance or directly using a PM at the cost p_j. Similarly, the demand of each user job i is also represented by a vector $R_i = (r_{i1}, r_{i2}, \ldots, r_{id})$ where r_{ik} is user i's demand on the dimension-k resource. We assume that both r_{ik} and c_{kj} are normalized to the maximum value of c_{kj} among all the types of deals, *i.e.,* $\max_{j\in\{1,\ldots,n\}} c_{kj} = 1$ for $\forall k \in \{1, ..., d\}$. Therefore, we have $0 < r_{ik} \leqslant 1$ and $0 < c_{kj} \leqslant 1$.

The objective of the provider is to minimize the total cost of all the group buying deals launched while satisfying all users' demands, by allocating each job to a proper deal. Let x_i^{js} be a binary job-to-deal allocation variable: if user i's job is allocated to the s^{th} type-j group buying deal, then $x_i^{js} = 1$, otherwise $x_i^{js} = 0$. Let y_{js} be a binary deal selection variable: if there is any user job allocated to the s^{th} type-j group buying deal, then $y_{js} = 1$, otherwise $y_{js} = 0$. Apparently, for each type of group buying deal, the number of deals used will be upper-bounded by the number of user jobs, *i.e.*, we have $s \in \{1, \ldots, m\}$.

The static group buying organization can be formulated as a bin packing problem in which the target is to pack a set of variable-sized items into a number of bins at a minimal cost. If we consider group buying deals as bins and user demands as items, the optimization problem we aim to solve in the static grouping strategy can be formulated as:

$$\min \quad \sum_{1\leqslant j\leqslant n} \sum_{1\leqslant s\leqslant m} p_j y_{js} \tag{1}$$

$$\text{s.t.} \quad \sum_{1\leqslant j\leqslant n} \sum_{1\leqslant s\leqslant m} x_i^{js} = 1, \quad i \in \{1, \ldots, m\}, \tag{2}$$

$$\sum_{1\leqslant i\leqslant m} r_{ik} x_i^{js} \leqslant c_{kj} y_{js},$$

$$j \in \{1, \ldots, n\},\ s \in \{1, \ldots, m\},\ k \in \{1, \ldots, d\}, \tag{3}$$

$$x_i^{js} \in \{0, 1\},$$

$$j \in \{1, \ldots, n\},\ i \in \{1, \ldots, m\},\ s \in \{1, \ldots, m\}, \tag{4}$$

$$y_{js} \in \{0, 1\}, \quad j \in \{1, \ldots, n\},\ s \in \{1, \ldots, m\}. \tag{5}$$

The objective function (1) minimizes the aggregate cost of selected group buying deals. Constraint (2) ensures that each user job is allocated to exactly one group buying deal. Constraint (3) ensures that a group buying deal will be selected when there are jobs allocated to it, and the resources offered in the deal can satisfy all the demand of user jobs allocated to it.

3.2. An Optimal Branch-and-Price Algorithm

Problem (1) is equivalent to the variable-sized vector bin packing problem, which is proved to be NP-hard. Fortunately, for a small number of jobs, we can use branch-and-price [Barnhart et al. 1998], a framework to solve generalized assignment problems in the literature, to obtain the optimal group organization. We now present a branch-

and-price algorithm particularly adapted to our problem (1). Specifically, we reformulate problem (1) into a master problem and a set of subproblems with Dantzig-Wolfe decomposition [Dantzig and Wolfe 1960]. We start with a heuristic initial solution, and improve the solution by solving the master problem and subproblems iteratively. Since the master problem is modeled as a linear programming problem, a branch-and-bound framework is utilized to restrict the solution into integer.

Dantzig-Wolfe decomposition: Before illustrating Dantzig-Wolfe decomposition, we introduce some new concepts. Note that multiple user demands are organized into a group and then allocated to a group buying deal. For a group of users, if their aggregate demands can be satisfied by a type-j group buying deal, we call this group a *feasible allocation* to type-j deals. Clearly, for each type of group buying deal, there are multiple feasible allocations.

With Dantzig-Wolfe decomposition, our static group organization problem is decomposed into a combination of a *allocation-choosing master problem* and multiple *allocation-generation subproblems*. In the allocation-choosing master problem, we are given a number of feasible allocations, from which we choose the proper ones to minimize the total group buying cost. Corresponding to each type of group buying deal, there is an allocation-generation subproblem. In these subproblems, we properly generate feasible allocations to each type of group buying deals. The optimal solution will be found through the interaction of the allocation-choosing problem and allocation-generation problems. In the following, we will formulate these master and subproblems, and show how they interact with each other.

Allocation-choosing master problem: We use $B_l^j = (b_{1l}^j, b_{2l}^j, \ldots, b_{ml}^j)$ to denote the l^{th} feasible allocation of type-j group buying deals. b_{il}^j is a binary variable which indicates whether job i is in B_l^j. We use q_j to denote the number of feasible allocations to type-j deals. We use λ_l^j to denote whether B_l^j is chosen. Note that in the master problem, λ_l^j is a continuous variable, which means that its solution could choose only a portion of a feasible allocation. We will show later how we use a branching strategy to restrict the solution to integer. After introducing the above variables, the allocation choosing problem can be formulated as:

$$\min \quad \sum_{1 \leqslant j \leqslant n} \sum_{1 \leqslant l \leqslant q_j} p_j \lambda_l^j \tag{6}$$

$$s.t. \quad \sum_{1 \leqslant j \leqslant n} \sum_{1 \leqslant l \leqslant q_j} b_{il}^j \lambda_l^j = 1, \quad i \in \{1, \ldots, m\}, \tag{7}$$

$$0 \leqslant \lambda_l^j \leqslant 1 \quad j \in \{1, \ldots, n\},\ l \in \{1, \ldots, q_j\}. \tag{8}$$

The master problem is modeled as a linear programming (LP) problem with an exponential number of variables, each corresponding to a feasible allocation. Even for a small number of jobs, the problem would be too large to be solved directly. To address this issue, we generate allocations dynamically by solving subproblems iteratively. In each iteration, the subproblems only generate allocations that have the potential to improve the existing solution. Hence, the master problem would have much fewer variables than the original problem. It can be solved efficiently by linear programming algorithms such as the simplex algorithm.

Allocation-generation subproblems: In each iteration there would be multiple subproblems, each corresponding to one type of group buying deals. The subproblem corresponding to type-j group buying deals is a multidimensional knapsack problem,

and it can be formulated as:

$$\max \quad \sum_{1 \leqslant i \leqslant m} u_i z_i - p_j \tag{9}$$

$$s.t. \quad \sum_{1 \leqslant i \leqslant m} r_{ik} z_i \leqslant c_{kj}, \quad k \in \{1, \ldots, d\}, \tag{10}$$

$$z_i \in \{0, 1\}. \tag{11}$$

In the subproblem, the parameters u_i are the dual variables of the solution to the allocation choosing problem. Applying the primal-dual theory, the value of u_i $(1 \leqslant i \leqslant m)$ can be calculated efficiently from the solution to the master problem, *i.e.*, λ_l^j, by solving the following equations:

$$\sum_{1 \leqslant i \leqslant m} u_i = \sum_{1 \leqslant j \leqslant n} \sum_{1 \leqslant l \leqslant q_j} p_j \lambda_l^j,$$

$$\sum_{1 \leqslant i \leqslant m} u_i b_{il}^j = p_j, \quad j \in \{1, \ldots, n\},\ l \in \{1, \ldots, q_j\}.$$

The solution $Z = (z_1, z_2, \ldots, z_m)$ is a new feasible allocation of type-j group buying deals. If Z can achieve a positive value of objective function (9), then Z is an allocation that has the potential to further improve the solution.

If feasible allocations with positive objective function values are found, a new iteration starts. The feasible allocations found in the last iteration will be added into the allocation choosing problem in the new iteration. With the results of allocation choosing problem, subproblems will be solved to find new feasible allocations. The algorithm progresses in such an iterative way until no feasible allocation can be found in any of the subproblems. The solution to the allocation choosing problem in the last iteration is then the optimal solution [Barnhart et al. 1998].

However, it is very likely that the solution is not integral. So we need a branching strategy to restrict the solution to integer when the solution is fractional. According to Ryan and Foster [Ryan and Foster 1981], if the solution is fractional, then we can always find a job pair i and i' which satisfy:

$$0 < \sum_{\forall l, j : b_{il}^j = 1, b_{i'l}^j = 1} \lambda_l^j < 1. \tag{12}$$

In this situation, we can generate a pair of branching constraints given by:

$$\sum_{\forall l, j : b_{il}^j = 1, b_{i'l}^j = 1} \lambda_l^j = 1, \quad \text{and} \tag{13}$$

$$\sum_{\forall l, j : b_{il}^j = 1, b_{i'l}^j = 1} \lambda_l^j = 0. \tag{14}$$

This branching scheme has a natural physical interpretation in the group buying problem. Eq. (12) indicates that in any fractional solution, there always exists such a pair of jobs i and i' that satisfies: 1) among all the deals chosen in the solution there is at least one deal that contains both i and i', and 2) in the meanwhile, there is also at least one deal which contains only one of the jobs. Accordingly, in the branching constraints, the branch corresponding to Eq. (13) requires that if a deal contains i, it should also contain i'. In other words, i and i' should always be allocated to the same group buying deals. The other branch, which corresponds to Eq. (14), requires that if an allocation

contains i, then it should not contain i' and vice versa. Therefore, i and i' are always required to be allocated to different deals. For any solution, if no such job pairs can be found, then it must be integer.

A branch-and-bound framework is used to handle branches. In each branch, a similar allocation generation method is used to solve the respective allocation choosing problem. The algorithm stops when the optimal integral solution is found. The efficiency of a branch-and-bound algorithm largely depends on the bounds of the LP relaxation problem at each node in the branch tree. In the branch-and-price algorithm, the LP bounds of the problem after Dantzig-Wolfe decomposition are quite tight [Dantzig and Wolfe 1960], thus limiting the branch tree to a small size. An algorithm with a smaller tree size would be more efficient, since there are fewer branches to explore. In Sec. 6, we will further evaluate the convergence speed of the branch-and-price algorithm via trace-driven simulation. The framework of our static group organization algorithm is presented in Algorithm 1.

ALGORITHM 1: Optimal Static group buying organization

```
1: Generate a initial set of feasible allocations with heuristic algorithm
2: Solve the allocation-choosing problem
3: Calculate dual variables
4: Solve the allocation-generation subproblems
5: if new feasible allocations are found then
6:     goto 2
7: end if
8: if the solution of restricted master problem is fractional then
9:     Branch and goto 2
10: end if
11: Stop
```

3.3. An Approximation Algorithm

Although the above-mentioned branch-and-price solution is optimal, in practice, it is impractical to organize group buying for a large batch of user jobs, since we require the solutions to both the master problem and subproblems to be integral. In particular, in the allocation generation subproblem, for each job, the algorithm needs to decide whether to add it into the generated allocation, therefore the computation complexity is $O(2^n)$. In the branching strategy of the master problem, for each pair of jobs there are two corresponding branches. Therefore, the algorithm needs to explore $O(2^{n^2})$ branches in the worst case.

Leveraging the branch-and-price framework, in *Cocoa*, we adopt an approximation algorithm in [Patt-Shamir and Rawitz 2012] to solve the static grouping problem in polynomial time. Although for both the master problem and the subproblem, calculating the integral solutions are proved to be NP-hard, their fractional solutions can be obtained in polynomial time. In *Cocoa*, we leverage the fractional solutions to get approximate integral solutions to the master problem and subproblems. In particular, to avoid the complexity of calculating the optimal integral solution to the subproblems, the algorithm exploits a polynomial time approximation scheme (PTAS) to generate feasible allocations. With the generated allocations, we can get the fractional solution to the master problem in polynomial time. From the fractional solution, we will derive an approximate integral solution via a greedy process, to avoid the complexity of exploring the branches in the branch-and-bound framework.

A PTAS of the subproblem. For each type of group buying deal, *Cocoa* applies the PTAS to the corresponding allocation generation subproblem (9). Specifically, for type-j group buying deals, to achieve an approximation of $1-\varepsilon$, the algorithm generates candidate allocations from all the subset $\mathbb{G}$ of jobs that satisfies $|\mathbb{G}| \leqslant q = \min\{m, \lceil d/\varepsilon \rceil\}$, where $\varepsilon > 0$ is an arbitrary real number. Recall that m is the number of user jobs and d is the dimension of resources. The algorithm discards any $\mathbb{G}$ that cannot be packed into a type-j group buying deal. For each remaining $\mathbb{G}$, we generate a candidate allocation that contains all jobs in $\mathbb{G}$. In particular, we first pack jobs in $\mathbb{G}$ into a type-j deal, and then pack some of the leftover jobs, whose coefficient (u_i) is no bigger than any job in $\mathbb{G}$, into the remaining deal capacity. We pack leftover jobs by solving the LP relaxation of a knapsack problem and rounding down all the fractional values. Jobs in $\mathbb{G}$ along with leftover jobs packed into the deal are considered as a candidate allocation. Among all candidate allocations generated from all the possible $\mathbb{G}$s, the algorithm chooses the one that maximizes the objective function (9) as the approximation solution. This process can achieve a $1-\varepsilon$ approximation to the optimal solution in polynomial time [Patt-Shamir and Rawitz 2012].

A greedy approximation for the master problem. Using PTAS for all the subproblems, we can get the approximate fractional solution to the master problem in polynomial time [Plotkin et al. 1995]. From the fractional solution, we calculate an approximation integral solution to the master problem in a greedy manner. Let $\mathcal{B}^+$ be the set of all the allocations with non-zero values in the fractional solution. $B_l'^j = (b_{1l}'^j, b_{2l}'^j, \ldots, b_{ml}'^j)$ denotes the l^{th} feasible allocation of type-j group buying deals in $\mathcal{B}^+$, where $b_{il}'^j$ is a binary variable indicating whether user job i is in $B_l'^j$. Let $\mathcal{G}$, which is initially empty, be the set of allocations finally chosen in the approximate solution. For all the feasible allocations in $\mathcal{B}^+$, the algorithm greedily chooses the one with largest $(\sum_{1\leqslant i\leqslant m} u_i b_{il}'^j)/p_j$ and adds it to $\mathcal{G}$. u_i is the dual variable corresponding to job i, which is derived from the solution to the master problem. The greedy scheme keeps progressing until $\sum_{B_l'^j \in \mathcal{G}} p_j < \ln 2d \cdot W^*$, where W^* is the value of the objective function (6) obtained from the approximate fractional solution to the master problem. For each feasible allocation $B_l'^j$ in $\mathcal{G}$, we open a type-j group buying deal and allocate all the jobs in $B_l'^j$ into the deal. For the residual jobs not contained in any feasible allocation in $\mathcal{G}$, the algorithm opens new group buying deals and packs them in a first-fit manner.

The approximation algorithm solves the static group buying problem in polynomial time with an approximation ratio of $\ln 2d+3$ in the worst case [Patt-Shamir and Rawitz 2012], where d is the number of resource types considered. In Sec. 6, we will show through simulations that the approximation algorithm almost achieves the same performance as the branch-and-price algorithm does, while saving running time by 60×.

4. HANDLE JOB DYNAMICS WITH AN ONLINE KNAPSACK ALGORITHM

Different from conventional VM-based workload consolidation, *Cocoa* uses a dynamic grouping strategy, in addition to the proposed static strategy, to cope with job arrivals and departures on the go with an online knapsack algorithm. Although the static group buying strategy can pack a batch of jobs into as few group buying deals as possible to minimize the total deal cost, jobs usually do not finish at the same time; resource utilization may degrade as some jobs depart from a deal, leading to resource "holes". Therefore, a dynamic strategy is needed to allocate incoming jobs to group buying deals to fill these resource "holes" caused by departed jobs. In this section, we first describe the framework we use in the dynamic grouping strategy to schedule user jobs. Then we formulate the problem to solve in the dynamic grouping strategy as an

online multidimensional knapsack problem. After that, we propose the learn-and-pack algorithm, which is designed to make dynamic grouping decisions. Later in Sec. 5, we will analyze the competitiveness of the proposed learn-and-pack algorithm.

For all the running group buying deals, *Cocoa* periodically eliminates containers with finished jobs and recycle the released resources to launch containers for newly arrived jobs. In practice, some cloud services also perform resource reorganization periodically. For example, in Amazon EC2 Spot instance, the provider updates the service price and reallocate cloud resources in every five-minute interval [Amazon EC2 Spot Instance Pricing 2014]. Considering the fluctuation of job arrival rate, in *Cocoa*, we adopt an adjustable recycle interval. We use short intervals during periods of high arrival rates to serve more jobs, and longer intervals for low arrival rates to reduce overhead. Specifically, we perform resource recycling whenever a predefined number of T jobs have arrived since the last recycle is performed.

The framework for dynamic grouping: Before illustrating the dynamic grouping problem, we first provide an overview of the dynamic scheduling framework for jobs and deals in *Cocoa*. Recall that in the static grouping strategy, *Cocoa* creates new group buying deals to pack jobs. We maintain a *running queue* for all the active group buying deals. Within each recycle interval, for each job that finishes during the interval, *Cocoa* eliminates its corresponding container at the end of interval so as to free up idle resources. If all the containers in a deal are eliminated, then we also eliminate the deal and remove it from the running queue. At the beginning of each recycle interval, we check each group buying deal in the running queue for its amount of idle resources. Upon the arrival of a new job, we use a learn-and-pack algorithm to decide whether to allocate it to the running deals and which deal it should be allocated to. The details of the learn-and-pack algorithm will be elaborated later in this section. Once a job is allocated to a deal, a container will be launched according to the resource demand of the job. Thanks to modern virtualization technology, the process of container creation and setup is efficient and the corresponding overhead is small. The users of *Cocoa* are willing to accept this overhead since their jobs are delay tolerant. *Cocoa* also maintains a buffer (*i.e.,* the waiting queue) for the jobs that are not packed into any running deals. Once the number of jobs in the waiting queue reaches a predefined threshold, *Cocoa* will use the static grouping strategy to pack them into new created deals. Recall that *Cocoa* avoids high queuing delay for jobs waiting in the queue by setting an upper bound for waiting time. When the waiting time of any job reaches the upper bound, *Cocoa* immediately performs the static grouping. For jobs that are dynamically grouped, they are allocated to a deal upon their arrival, thus will not have a queuing delay.

Problem Formulation: each recycle interval, if we consider each group buying deal with idle resources as a knapsack and each incoming user job as an item, the dynamic grouping problem can be formulated as an multi-dimensional multiple knapsack problem, where new user jobs are packed into resource "holes" with an objective of maximizing the resource utilization at the end of the recycle interval. Assume there are H group buying deals with the vector $C_h = (c_{1h}, c_{2h}, \ldots, c_{dh})$ denoting the amount of idle resources in the h^{th} deal in all d resource dimensions. Let S be the set of jobs arrived in the interval and $R_i = (r_{i1}, r_{i2}, \ldots, r_{id})$ be the resource demand vector of i^{th} job. For resource dimension k, we assume that both r_{ik} and c_{kh} are normalized to the largest value of c_{kh} for all h from all the intervals, thus $0 < r_{ik} \leqslant 1$ and $0 < c_{kh} \leqslant 1$. The

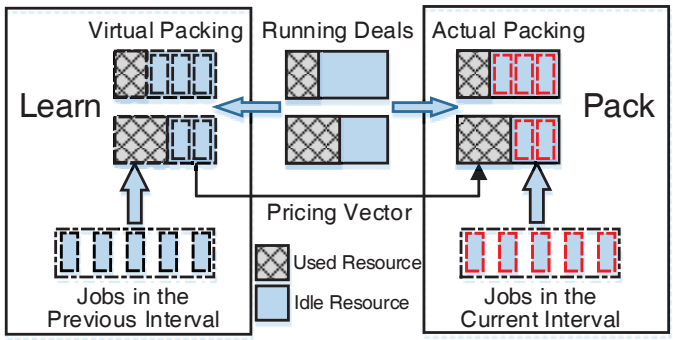

Fig. 3. The framework of the learn-and-pack strategy. At the beginning of each interval *Cocoa* learns the pricing vector of each running deal by virtually packing jobs in the previous interval to the deals. The learnt pricing vector is used to pack new coming jobs in a online manner.

problem we aim to solve in the dynamic grouping strategy can be formulated as:

$$\max \quad \sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i x_{ih} \tag{15}$$

$$\text{s.t.} \quad \sum_{i\in S} r_{ik}x_{ih} \leqslant c_{kh},$$

$$k \in \{1,\ldots,d\},\ h \in \{1,\ldots,H\}, \tag{16}$$

$$\sum_{1\leqslant h\leqslant H} x_{ih} \leqslant 1, \quad i \in S, \tag{17}$$

$$x_{ih} \in \{0,1\}, \quad i \in S. \tag{18}$$

where π_i is the value earned by the provider if job i is packed into any running deal. Recall that the objective of the dynamic grouping is to maximize the utilization of idle resource in running deals so as to maximize the accommodated resource requirements from newly arrived jobs. Therefore, π_i should be interpreted as the resource demand of job i. Considering multiple resource dimensions, we calculate π_i as a weighted sum of job i's resource usage in all resource dimensions. In practice, the provider can set the weight of each resource dimension according to its price or resource scarcity. In this paper, we set all the weights of resource dimensions equal to one for simplicity. Our dynamic grouping strategy works as long as π_i is a weighted sum of job i's resource usage in all resource dimensions.

Constraint (16) ensures that the total resource demand of jobs allocated to a deal is no bigger than its capacity of idle resources. Constraint (17) ensures that a job is allocated to at most one running deal. In *Cocoa*, we deal with the online version of problem (15). Upon the arrival of each incoming job, a carefully designed online strategy is adopted to make proper decisions on whether to pack the incoming job into any running deals and which deal to pack it into.

A Learn-and-Pack Strategy: We propose an online algorithm for the knapsack problem to pack each new job in a recycle interval into a proper deal by learning from a *virtual offline packing problem* of the jobs in the previous interval. The framework of the learn-and-pack strategy is presented in Fig. 3. For the t^{th} recycle interval, to learn from jobs in the $(t-1)^{th}$ interval, we consider a *virtual* offline knapsack problem, which is a linear program that *virtually* packs all jobs in the $(t-1)^{th}$ interval into the idle resources in the present running group buying deals. Let $\widehat{S}$ be the set of user jobs that arrived at the $(t-1)^{th}$ interval. Let $\widehat{R}_i = (\widehat{r}_{i1}, \widehat{r}_{i2}, \ldots, \widehat{r}_{id})$ be the resource demand vector of i^{th} job. For resource dimension k, we assume that $\widehat{r}_{ik}$ is normalized

to the largest value of c_{kh} for all h from all the intervals, thus $0 < \widehat{r}_{ik} \leqslant 1$. The virtual knapsack problem is formulated as:

$$\max \quad \sum_{i \in \widehat{S}} \sum_{1 \leqslant h \leqslant H} \widehat{\pi}_i \widehat{x}_{ih} \tag{19}$$

$$\text{s.t.} \quad \sum_{i \in \widehat{S}} \widehat{r}_{ik} \widehat{x}_{ih} \leqslant c_{kh},$$

$$k \in \{1, \ldots, d\},\ h \in \{1, \ldots, H\}, \tag{20}$$

$$\sum_{1 \leqslant h \leqslant H} \widehat{x}_{ih} \leqslant 1, \quad i \in \widehat{S}, \tag{21}$$

$$\widehat{x}_{ih} \geqslant 0, \quad i \in \widehat{S}. \tag{22}$$

The objective function (19) maximizes the total value (resource demand) of jobs that are packed into existing deals. Constraint (20) ensures that the capacity of idle resources in a deal is no smaller than the demand of jobs allocated to it. Note that the problem is a linear program, one job can be allocated to multiple deals, with each deal accommodating a portion of it. Constraint (21) ensures that for each job the sum of all the portions allocated to existing deals is no more than one, *i.e.,* a job is allocated for at most once. Constraint (22) is a relaxation of the integer constraint in the knapsack problem, it allows the solution $(\widehat{x}_{ih})$ to be fractional.

Consider the dual problem of (19), which can be formulated as the following:

$$\min \quad \sum_{1 \leqslant k \leqslant d} \sum_{1 \leqslant h \leqslant H} c_{kh} \widehat{p}_{kh} + \sum_{i \in \widehat{S}} \widehat{y}_i \tag{23}$$

$$\text{s.t.} \quad \sum_{1 \leqslant k \leqslant d} \widehat{r}_{ik} \widehat{p}_{kh} + \widehat{y}_i \geqslant \widehat{\pi}_i,$$

$$i \in \widehat{S},\ h \in \{1, \ldots, H\}, \tag{24}$$

$$\widehat{p}_{kh}, \widehat{y}_i \geqslant 0. \tag{25}$$

The set of dual variables $\widehat{P}_h = (\widehat{p}_{1h}, \widehat{p}_{2h}, \ldots, \widehat{p}_{dh})$ (*i.e.*, the pricing vector) corresponding to deal h can be used to indicate whether a newly arrived job in the t^{th} interval should be allocated to group buying deal h. $\widehat{p}_{kh}$ can be interpreted as the unit price of dimension-k idle resource in deal h in the virtual packing problem. In the learn-and-pack strategy we estimate the cost of packing the newly arrived job i into group buying deal h as $\sum_{1 \leqslant k \leqslant d} r_{ik} \widehat{p}_{kh}$. For each deal h, the utility of allocating job i to deal h is calculated as $w_{ih} = \pi_i - \sum_{1 \leqslant k \leqslant d} r_{ik} \widehat{p}_{kh}$. In *Cocoa*, we allocate job i to the deal it can fit into and achieves the largest utility. If there is no deal that can either achieve a positive utility or able to pack the job, then we add the job to the tail of the waiting queue. We will illustrate later in Sec. 5 that for any $\epsilon > 0$ the learn-and-pack algorithm can achieve $1 - 5\epsilon$ competitive to the optimal offline algorithm.

A Prioritized Learn-and-Pack Strategy: Unfortunately, the learn-and-pack strategy may still result in resource inefficiency, when the total user demand is constantly decreasing. When the total demand decreases, all the deals will have "holes". In this case, the system should scale down by flushing and removing some of the deals, while keeping other deals running at a high level of utilization. However, in the above learn-and-pack strategy, the incoming jobs have similar chances to be packed into all the deals. When the total demand decreases, the number of jobs in each deal is decreasing at a similar rate. In the end, all the deals will have a low utilization with only a few jobs running.

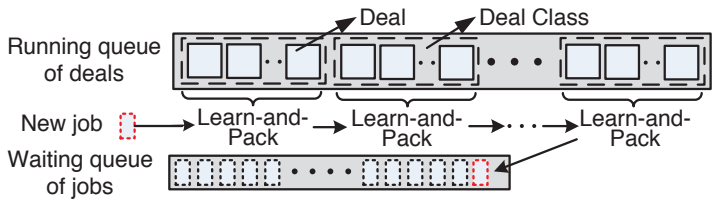

Fig. 4. When a new job comes, *Cocoa* tries to allocate it to running deals class by class. The learn-and-pack strategy is used within each class. If there is no deal that a job can be allocated to, it is added to the tail of the waiting queue.

As shown in Fig. 4, in *Cocoa*, we use a dynamic grouping strategy that complements the learn-and-pack strategy by prioritizing group buying deals. Specifically, we further divide deals in the running queue into classes of a fixed size. When a new job arrives, we check a class which is closer to the head of the running queue with a higher priority. Within each class, a learn-and-pack strategy is used to allocate the job to the proper deal. If the new job is not packed into any deal in the class, then the next class is checked. In this way, when the demand decreases, the deals in classes near the head of the queue have a higher priority to accommodate new jobs, and thus can maintain higher resource utilization. In the meantime, the deal classes near the tail will receive fewer incoming jobs and will eventually be eliminated as jobs leave.

Deal Management in the Running Queue: *Cocoa* organizes deals in the running queue in the form of ordered classes. We move running deals among classes in two situations: when new deals are created or existing deals are eliminated. *Cocoa* places newly created deals into the first class in the running queue. To make room for the newly created deals, we will move existing deals with lower utilization from the first class to subsequent classes. Recall that *Cocoa* performs static grouping to create new deals and pack jobs in the waiting queue into them. Since the static grouping strategy is near-optimal, new deals often have a high resource utilization, and thus are not likely to be freed up soon. Hence, we place new deals into the first class. On the other hand, when a deal is freed up, we move the deal with the highest utilization in the subsequent class to the current class, so as to maintain the predefined number of deals in the class.

5. COMPETITIVE ANALYSIS OF THE LEARN-AND-PACK ALGORITHM

In this section, we analyze the competitiveness of the learn-and-pack algorithm. Specifically, we prove that under the random permutation model, for any $\epsilon > 0$, the learn-and-pack algorithm can obtain a solution that is at least $1-5\epsilon$ times of the value of the optimal offline solution.

The permutation model is widely adopted in the competitiveness analysis of online algorithms [Ho and Vaughan 2012], [Mahdian and Yan 2011], [Goel and Mehta 2008]. It assumes that new jobs arrive in a random order, which is uniformly distributed over all job arrive orders. It is worth noting that the above competitive ratio also applies to the case where the resource demand of all jobs are drawn independently from some unknown distribution, *i.e.*, when the strategy is used to pack $i.i.d$ jobs into group buying deals.

THEOREM 5.1. *For any $\epsilon > 0$ that satisfies (26), the learn-and-pack algorithm is $1-5\epsilon$ competitive to the optimal offline algorithm in the random permutation model for all inputs, where*

$$C = min_{h,k} c_{kh} \geqslant \frac{4dHlog(T/\epsilon)}{\epsilon^2}. \tag{26}$$

Table I. List of Major Notations

H	number of running deals
T	number of incoming jobs in each recycle interval
S	the set of jobs in the current recycle interval
π_i	the value of job i
C_h	the vector of idle resource capacity in deal h
x_{ih}	a solution to the offline packing problem (15)
x^*_{ih}	the optimal solution to the offline packing problem (15)
$\widetilde{x}^*_{ih}$	the optimal solution to the LP relaxation of the offline packing problem (15)
P_h	dual solution (price vector) corresponding to deal h
$\widetilde{P}^*_h$	the optimal dual solution corresponding to deal h in the LP relaxation of problem (15)
$x_{ih}(P)$	solution to problem (15) under pricing vector P without considering deal capacity constraint
$x^f_{ih}(P)$	solution to problem (15) under pricing vector P in the learn-and-pack strategy
$\widehat{S}$	the set of jobs in the previous recycle interval
$\widehat{P}_h$	dual solution corresponding to deal h learnt in the virtual packing problem
$\widehat{x}_{ih}(\widehat{P})$	solution to problem (19) under pricing vector $\widehat{P}$ without considering deal capacity constraint
$\widehat{x}^f_{ih}(\widehat{P})$	solution to problem (19) under pricing vector $\widehat{P}$ in the learn-and-pack strategy

In (26) T is the number of incoming jobs in each recycle interval. Let $x^f_{ih}(\widehat{P})$ be the solution of the learn-and-pack strategy using the pricing vectors $\widehat{P}$ obtained from the virtual packing problem. Then Theorem 5.1 implies that the expectation of the solution value under the learn-and-pack strategy is at least $1-5\epsilon$ times of the solution value under the optimal offline solution:

$$E\Big[\sum_{i\in S}\sum_{1\leqslant h\leqslant H}\pi_i x^f_{ih}(\widehat{P})\Big] \geqslant (1-5\epsilon)OPT. \tag{27}$$

We first present an overview of the steps we take to prove Theorem 5.1. In Lemma 5.2 and Lemma 5.3, we assume that the optimal dual solution $\widetilde{p}^*_{kh}$ to LP relaxation of the offline packing problem (15) is known, and prove that learning from the dual solutions is adequate to obtain a near-optimal online solution. However, in *Cocoa*, decisions on dynamic job packing are made in an online manner, and the dual solutions cannot be acquired before decision making. Therefore, in the learn-and-pack strategy, we use the dual solutions in the virtual packing problem (19). In Lemma 5.4, we prove that, by using the dual solutions in the virtual packing problem (19), the learn-and-pack strategy can obtain a solution close to the optimal offline solution, with a high probability. After that, we will illustrate how to get the claim in Theorem 5.1 from Lemma 5.4.

Next, we introduce some concepts and definitions. Let $\widetilde{x}^*_{ih}$ denote the optimal solution to the linear programming relaxation of the offline actual packing problem (15). Let $x^f_{ih}(P)$ be the solution of the learn-and-pack strategy using a given pricing vector P. Note that $P=(P_1,P_2,...,P_H)$ with each P_h corresponds to the pricing vector of the running deal h. We also consider another online packing strategy which does not consider the capacity of running deals and simply allocate jobs to the deal that generates the maximal positive utility. We denote the solution of this strategy as $x_{ih}(P)$. Note that $x_{ih}(P)$ might not be a feasible solution, since it may violate the deal capacity constraints.

LEMMA 5.2. *For the LP relaxation of the offline packing problem (15), given the optimal pricing vector $\widetilde{P}^*$, if a job is packed in the solution of the online strategy that does not consider the capacity constraint $\big(x_{ih}(\widetilde{P}^*)\big)$, it will also be packed in the optimal*

offline solution $(\widetilde{x}^*_{ih})$. *Furthermore, there are at most* $d*H$ *jobs which are packed in* $\widetilde{x}^*_{ih}$, *while not packed in* $x_{ih}(\widetilde{P}^*)$. *In other words,* $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$ *for all* i *and* $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) \neq \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$ *by no more than* $d*H$ *values of i.*

PROOF. For a new coming job i, we consider two possible cases. For the first case where $(\widetilde{P}^*_h)^T R_i \neq \pi_i$ for all h, we prove that $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$. For the second case where there exists at least one h such that $(\widetilde{P}^*_h)^T R_i = \pi_i$, we prove that $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$. After that, we show that $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by no more than $d*H$ values of i.

According to the complementarity slackness conditions in the primal-dual theory, for the optimal solution $\widetilde{x}^*_{ih}$ to the LP relaxation of primal problem (15) and optimal solution $(\widetilde{p}^*_{kh}, \widetilde{y}^*_i)$ to the dual, we have:

$$\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} \prod_{1\leqslant h\leqslant H} \Big(\sum_{1\leqslant k\leqslant d} \widetilde{p}^*_{kh} r_{ik} + \widetilde{y}^*_i - \pi_i \Big) = 0, \tag{28}$$

$$(1 - \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}) \widetilde{y}^*_i = 0. \tag{29}$$

For the first case where $(\widetilde{P}^*_h)^T R_i \neq \pi_i$ for all h, we consider two possible situations. In the first situation, *i.e.,* $(\widetilde{P}^*_h)^T R_i > \pi_i$ for all h, we have $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = 0$, since job i will not be packed if no deal can achieve a positive utility. According to (28), we also have $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} = 0$. Therefore, we have $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} = \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$. In the second situation, where there exists some h such that $(\widetilde{P}^*_h)^T R_i < \pi_i$, there must be some h' that satisfies $x_{ih'}(\widetilde{P}^*) = 1$. This is because, when deal capacity is not considered, a job will eventually be allocated to a deal if it can achieve positive utility. Therefore, we have $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = 1$. By constraint (24) and the complementarity condition (29), we also have $\widetilde{y}^*_i > 0$ and $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} = 1$. Combining the conclusions in the two possible situations, we have $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} = \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ in the first case.

For the second case where there exists at least one h such that $(\widetilde{P}^*_h)^T R_i = \pi_i$, if there exists some h such that $(\widetilde{P}^*_h)^T R_i < \pi_i$, similar to the second situation in the first case, we have $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$. Otherwise, if $(\widetilde{P}^*_h)^T R_i \geqslant \pi_i$ for all h, then we have $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$, since $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = 0$ and $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} \geqslant 0$. Therefore, we have $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} \leqslant \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ in the second case.

Note that $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih} \neq \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ only in the second case, *i.e.,* when there exists some h such that $(\widetilde{P}^*_h)^T R_i = \pi_i$. According to [Devanur and Hayes 2009], we can safely assume that for each running deal h there are at most d values of i such that $(\widetilde{P}^*_h)^T R_i = \pi_i$, with arbitrarily small impact on the solution. Since there are H running deals, we have at most $d*H$ values of i such that $(\widetilde{P}^*_h)^T R_i = \pi_i$ for some h. Then we can get the conclusion that $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by no more than $d*H$ values of i. □

LEMMA 5.3. *For the LP relaxation of the offline packing problem (15), given the optimal pricing vector* $\widetilde{P}^*$, *if a job is packed in the solution of the learn-and-pack strategy* $\big(x^f_{ih}(\widetilde{P}^*)\big)$, *it will also be packed in the optimal offline solution* $(\widetilde{x}^*_{ih})$. *Furthermore, there are at most* $2d*H$ *jobs which are packed in* $\widetilde{x}^*_{ih}$, *while not packed in* $x^f_{ih}(\widetilde{P}^*)$. *In other*

words, $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} \widetilde{x}_{ih}^*$ *for all* i *and* $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*) \neq \sum_{1\leqslant h\leqslant H} \widetilde{x}_{ih}^*$ *by no more than* $2d * H$ *values of* i*.*

PROOF. It is obvious that $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$, since every job i packed in the learn-and-pack strategy will also be packed in the strategy that do not consider deal capacity. Combined with the conclusion in Lemma 5.2, we have $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H} \widetilde{x}_{ih}^*$.

In the following, we show that $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*)$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by no more than $d * H$ values of i. First, define $S_d = \{i | \sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*) = 1, \sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*) = 0\}$, as the set of jobs which are packed in the strategy that does not consider capacity constraints while are not packed in the learn-and-pack strategy. Let $lf_{kh} = c_{kh} - \sum_{i\in S} r_{ik} x_{ih}^f(\widetilde{P}^*)$, be the left capacity in resource dimension k of deal h in learn-and-pack strategy, at the end of the recycle interval. Since for each job i in set S_d, there is no group buying deal it can fit into, we have for $\forall h \in \{1,\ldots,H\}, \forall i \in S_d, \exists k \in \{1,\ldots,d\}$ such that $r_{ik} > lf_{kh}$. Considering all the jobs in S_d, there must exist at least one resource dimension, denoted as dimension κ, such that for running deal h the number of jobs that satisfies $r_{i\kappa} > lf_{\kappa h}$ is at least $|S_d|/d$, where d is the number of considered resource dimensions.

Next, our proof proceeds using the method of reduction to absurdity. In particular, let us assume that $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*)$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by more than $d * H$ values, *i.e.*, $|S_d|/d > H$. Then, for resource dimension κ there are at least H jobs that satisfy $r_{i\kappa} > lf_{\kappa h}$. Therefore, corresponding to each deal h, we can have a unique job $i(h) \in S_d$, such that $r_{i(h)\kappa} > lf_{\kappa h}$. Thus, we have:

$$\sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} x_{ih}^f(\widetilde{P}^*) + \sum_{1\leqslant h\leqslant H} r_{i(h)\kappa} = \sum_{1\leqslant h\leqslant H} \big(\sum_{i\in S} r_{i\kappa} x_{ih}^f(\widetilde{P}^*) + r_{i(h)\kappa}\big) \tag{30}$$

$$> \sum_{1\leqslant h\leqslant H} \big(\sum_{i\in S} r_{i\kappa} x_{ih}^f(\widetilde{P}^*) + lf_{\kappa h}\big) \tag{31}$$

$$= \sum_{1\leqslant h\leqslant H} c_{\kappa h}. \tag{32}$$

According to the definition of S_d, we have

$$\sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} x_{ih}^f(\widetilde{P}^*) + \sum_{1\leqslant h\leqslant H} r_{i(h)\kappa} < \sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} x_{ih}^f(\widetilde{P}^*) + \sum_{i\in S_d} r_{i\kappa} = \sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} x_{ih}(\widetilde{P}^*). \tag{33}$$

According to Lemma 5.2, we also have

$$\sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} x_{ih}(\widetilde{P}^*) \leqslant \sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} \widetilde{x}_{ih}^*. \tag{34}$$

Then, we get to the following conclusion:

$$\sum_{1\leqslant h\leqslant H}\sum_{i\in S} r_{i\kappa} \widetilde{x}_{ih}^* > \sum_{1\leqslant h\leqslant H} c_{\kappa h}. \tag{35}$$

Eq. (35) implies that $\widetilde{x}_{ih}^*$ is not a feasible solution to the LP relaxation of the offline packing problem (15). This is in conflict with $\widetilde{x}_{ih}^*$'s definition as the optimal solution to the LP relaxation of problem (15). Therefore, the assumption that $\sum_{1\leqslant h\leqslant H} x_{ih}^f(\widetilde{P}^*)$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by more than $d*H$ values leads to an incorrect conclusion. Then,

we get the conclusion that $\sum_{1\leqslant h\leqslant H} x^f_{ih}(\widetilde{P}^*)$ and $\sum_{1\leqslant h\leqslant H} x_{ih}(\widetilde{P}^*)$ differ by no more than $d*H$ values of i. Combined with the result of Lemma 5.2, we have $\sum_{1\leqslant h\leqslant H} x^f_{ih}(\widetilde{P}^*)$ and $\sum_{1\leqslant h\leqslant H} \widetilde{x}^*_{ih}$ differ by no more than $2d*H$ values of i. □

LEMMA 5.4. *For the learn-and-pack strategy with pricing vector $\widehat{P}$ learnt from jobs in the previous recycle interval, with probability $1-\epsilon$, the solution of is near-optimal:*

$$\sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i x^f_{ih}(\widehat{P}) \geqslant (1-4\epsilon)OPT.$$

given $C \geqslant \frac{4dHlog(T/\epsilon)}{\epsilon^2}$.

PROOF.

Recall that $\widehat{P}$ is the pricing vector learnt by virtually packing jobs in the previous interval into the current running deals. We use $\widehat{x}^f_{ih}(\widehat{P})$ to denote the obtained solution to the virtual packing problem using the learn-and-pack strategy and pricing vector $\widehat{P}$. Note that $\widehat{x}^f_{ih}(\widehat{P})$ is a binary solution though the virtual packing problem is a linear program. Consider the virtual packing problem (19), since $\widehat{x}_{ih}$ is the optimal primal solution and $\widehat{P}$ is the corresponding dual solution, by the complementarity conditions of the linear program, if $\widehat{p}_{kh} > 0$, there will be $\sum_{i\in\widehat{S}} \widehat{r}_{ik}\widehat{x}_{ih} = c_{kh}$, where $\widehat{S}$ is the set of jobs that arrive in the $(t-1)^{th}$ interval. Recall that $0 < \widehat{r}_{ik} \leqslant 1$, by applying Lemma 5.3 to the virtual packing problem, if $\widehat{p}_{kh} > 0$, we have

$$\sum_{i\in\widehat{S}} \widehat{r}_{ik}\widehat{x}^f_{ih}(\widehat{P}) \geqslant \sum_{i\in\widehat{S}} \widehat{r}_{ik}\widehat{x}_{ih} - 2d*H = c_{kh} - 2d*H. \tag{36}$$

Then by the constraint on C in Lemma 5.4, we further have

$$\sum_{i\in\widehat{S}} \widehat{r}_{ik}\widehat{x}^f_{ih}(\widehat{P}) \geqslant c_{kh} - 2d*H \geqslant c_{kh} - \frac{\epsilon}{2log(T/\epsilon)}\epsilon C \geqslant c_{kh} - \epsilon C \geqslant (1-\epsilon)c_{kh}. \tag{37}$$

Next, we fix $\widehat{p}_{kh}$, k and h and show that, in the permutation model, when $\widehat{p}_{kh} > 0$ the possibility that $\sum_{i\in S} r_{ik}x^f_{ih}(\widehat{P}) \leqslant (1-4\epsilon)c_{kh}$ is no bigger than $\frac{\epsilon}{dHT^{Hd}}$. First, define $Y_{ih} = r_{ik}x^f_{ih}(\widehat{P})$ if $i \in S$, and $Y_{ih} = \widehat{r}_{ik}\widehat{x}^f_{ih}(\widehat{P})$ if $i \in \widehat{S}$. $Z_{ih} = \frac{(2-4\epsilon)c_{kh}Y_{ih}}{\sum_{i\in S} Y_{ih}}$. We have

$$\begin{aligned}
&P\Big(\sum_{i\in\widehat{S}} Y_{ih} \geqslant (1-\epsilon)c_{kh}, \sum_{i\in\{\widehat{S}\cup S\}} Y_{ih} \leqslant (2-4\epsilon)c_{kh}\Big)\\
=&P\Big(\Big|\sum_{i\in\widehat{S}} Y_{ih} - \frac{1}{2}\sum_{i\in\{\widehat{S}\cup S\}} Y_{ih}\Big| \geqslant \epsilon c_{kh}, \sum_{i\in\{\widehat{S}\cup S\}} Y_{ih} \leqslant (2-4\epsilon)c_{kh}\Big)\\
\leqslant&P\Big(\Big|\sum_{i\in\widehat{S}} Y_{ih} - \frac{1}{2}\sum_{i\in\{\widehat{S}\cup S\}} Y_{ih}\Big| \geqslant \epsilon c_{kh}, \sum_{i\in\{\widehat{S}\cup S\}} Z_{ih} \leqslant (2-4\epsilon)c_{kh}\Big)\\
\leqslant&P\Big(\Big|\sum_{i\in\widehat{S}} Z_{ih} - \frac{1}{2}\sum_{i\in\{\widehat{S}\cup S\}} Z_{ih}\Big| \geqslant \epsilon c_{kh}\Big|\sum_{i\in\{\widehat{S}\cup S\}} Z_{ih} = (2-4\epsilon)c_{kh}\Big).
\end{aligned}$$

In the permutation model, we can regard $Z_{ih}, i \in \widehat{S}$ as a sample from $Z_{ih}, i \in \{\widehat{S}\cup S\}$ without replacement. According to the Hoeffding-Bernstein's inequiality for sampling

without replacement [van der Vaart and Wellner 1996], we further have:

$$P\Big(\Big|\sum_{i\in\widehat{S}} Z_{ih} - \frac{1}{2}\sum_{i\in\{\widehat{S}\cup S\}} Z_{ih}\Big| \geqslant \epsilon c_{kh}\Big| \sum_{i\in\{\widehat{S}\cup S\}} Z_{ih} = (2-4\epsilon)c_{kh}\Big) \leqslant 2exp\Big(-\frac{\epsilon^2 c_{kh}}{2-3\epsilon}\Big) \leqslant \frac{\epsilon}{dHT^{Hd}}. \tag{38}$$

The last inequality is because of the constraint made on C. Note that Eq. (38) is obtained for a fixed combination of $\widehat{p}_{kh}$, k and h. By summing over all the T^{Hd} distinct prices [Orlik and Terao 1992], $k \in \{1, ..., d\}$ and $h \in \{1, ..., H\}$, we have the conclusion that for all k and h, if $\widehat{p}_{kh} > 0$ then $\sum_{i\in\{\widehat{S}\cup S\}} Y_{ih} \geqslant (2-4\epsilon)c_{kh}$ with a probability no smaller than $1-\epsilon$.

Since $\widehat{x}^f_{ih}(\widehat{P})$ is a feasible solution to the virtual packing problem (19), we have $\sum_{i\in\widehat{S}} Y_{ih} = \sum_{i\in\widehat{S}} r_{ik}x^f_{ih}(\widehat{P}) \leqslant c_{kh}$. Therefore for all k and h, if $\widehat{p}_{kh} > 0$ then we have:

$$\begin{aligned} P\Big(\sum_{i\in S} r_{ik}x^f_{ih}(\widehat{P}) \geqslant (1-4\epsilon)c_{kh}\Big) &\geqslant P\Big(\sum_{i\in\{\widehat{S}\cup S\}} Y_{ih} \geqslant (2-4\epsilon)c_{kh}, \sum_{i\in\widehat{S}} Y_{ih} \leqslant c_{kh}\Big) \\ &\geqslant (1-\epsilon). \end{aligned} \tag{39}$$

With the above conclusion, we can construct the following linear program:

$$\max \quad \sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i x'_{ih} \tag{40}$$

$$\text{s.t.} \quad \sum_{i\in S} r_{ik}x'_{ih} \leqslant c'_{kh},$$

$$k \in \{1, ..., d\},\ h \in \{1, ..., H\}, \tag{41}$$

$$0 \leqslant x'_{ih} \leqslant 1,\ i \in S, \tag{42}$$

$$0 \leqslant \sum_{1\leqslant h\leqslant H} x'_{ih} \leqslant 1,\ i \in S. \tag{43}$$

where $c'_{kh} = \sum_{i\in S} r_{ik}x^f_{ih}(\widehat{P})$ if $\widehat{p}_{kh} > 0$, and $c'_{kh} = max\{\sum_{i\in S} r_{ik}x^f_{ih}(\widehat{P}), c_{kh}\}$ if $\widehat{p}_{kh} = 0$. In the constructed problem (40), we aim to pack jobs in the current interval into deals with resource capacity $C'_h = (c'_{1h}, c'_{2h}, \ldots, c'_{dh}), 1 \leqslant h \leqslant H$. It is worth noting that problem (40) corresponds to neither the virtual packing problem nor actual packing problem. We construct it only to prove the near-optimality of the learn-and-pack strategy. According to the definition of c'_{kh}, we can get the conclusion that $c'_{kh} \geqslant (1-4\epsilon)c_{kh}$ with a probability no smaller than $1-\epsilon$ for all h and k. In addition, $x^f_{ih}(\widehat{P})$ and $\widehat{P}$ are the optimal primal and dual solution to linear program (40), since they satisfy all complementarity conditions.

With a probability no smaller than $1-\epsilon$, $c'_{kh} \geqslant (1-4\epsilon)c_{kh}$ holds for all h and k. Given x^*_{ih} as the optimal solution to the offline actual packing problem (15), $(1-4\epsilon)x^*_{ih}$ will be a feasible solution to problem (40). Since $x^f_{ih}(\widehat{P})$ is the optimal solution to problem (40), we have

$$\sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i x^f_{ih}(\widehat{P}) \geqslant \sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i(1-4\epsilon)x^*_{ih} \geqslant (1-4\epsilon)OPT. \tag{44}$$

□

According to Lemma 5.4, with probability $1-\epsilon$, we have:

$$\sum_{i\in S}\sum_{1\leqslant h\leqslant H} \pi_i x^f_{ih}(\widehat{P}) \geqslant (1-4\epsilon)OPT.$$

Denote this event by ε, where $P(\varepsilon) \geqslant 1-\epsilon$. We can get expected solution value of the learn-and-pack strategy:

$$E\Big[\sum_{i\in S}\sum_{1\leqslant h\leqslant H}\pi_i x_{ih}^f(\widehat{P})\Big] \geqslant (1-4\epsilon)P(\varepsilon)OPT \geqslant (1-5\epsilon)OPT. \quad (45)$$

Then we can have the conclusion in Theorem 5.1 that the learn-and-pack algorithm is $1-5\epsilon$ competitive to the optimal offline algorithm.

6. PERFORMANCE EVALUATION

We perform extensive trace-driven simulations based on a large amount of real-world traces to evaluate the performance of the proposed group buying organization strategies. In our simulation, we use the Google cluster workload trace data, which are publicly available [C. Reiss and Hellerstein 2011]. The trace set contains the information of resource usage in a Google cluster of 12,583 physical servers, which was collected during a period of 29 days in May 2011.

6.1. Dataset Preprocessing

The dataset records the resource usage of 933 Google users in the cluster. Users submit their demands as *jobs*. Each job consists of several tasks, each of which has a demand on computing resources such as CPU and memory. In our simulations, we deem each job in the dataset as a cloud user and organize the jobs into groups to perform group buying. We calculate the resource demand of a job by summing up the demands of all its tasks. We collect the information of resource demand, submission time, and completion time of jobs during the first week in the trace.

In the trace there are $89,768$ submitted jobs that are successfully executed in the first week. The distribution of jobs is highly skewed in terms of resource demand and lifetime. There are a few jobs with large demand while the majority have moderate demands. The distribution of job lifetimes has a similar pattern. Since *Cocoa* aims to organize small and short jobs into groups, we rule out jobs with large demands (the normalized demand larger than 0.1 in either CPU or memory) and long durations (longer than 900 seconds). In the trace, there is a "scheduling class" attribute corresponding to each job to specify its tolerance to service delay with a range from 0 to 3. Since *Cocoa* is designed to serve delay tolerant jobs, we remove delay-sensitive jobs with the value of "scheduling class" larger than 2. Note that in the Google trace, a moderate service delay is acceptable even for jobs whose "scheduling class" value are 3. There is also a "different-machine constraint" attribute to indicate whether two tasks in a job can be allocated to the same machine. In *Cocoa*, we allocate tasks of a job into one container. Therefore we remove those jobs whose tasks are required to be allocated to different machines in the trace. After preprocessing, we get $52,183$ remaining jobs in our simulation.

Since the resource usage in the trace is normalized by the amount of resources in the largest Google physical server. We assume the biggest server have 32 CPU cores with 128GB memory. Accordingly, we can calculate the actual resource demand of each job, shown in Fig. 5. The demand still has a skewed distribution after removing large jobs. In addition, there exists both CPU and memory intensive jobs, which calls for a careful consolidation scheme.

6.2. Group Buying Deal Setup

We define 5 types of group buying deals shown in Table II. Among them, there are two CPU/memory enhanced deals (*i.e.*, CPU/memory small and CPU/memory large), which offers a larger amount of resource in CPU/memory. To set the price of each

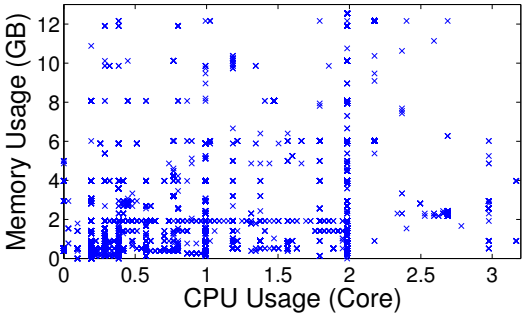

Fig. 5. The distribution of user demands for CPU and memory resources in the Google cluster workload traces.

Table II. Group buying deal settings

Name	CPU resource	Memory resource	Cost
CPU Large	12.8 cores	12.8GB	0.57$/hr
CPU Small	4.8 cores	9.6GB	0.24$/hr
Balanced	6.4 cores	25.6GB	0.42$/hr
Memory Small	3.2 cores	19.2GB	0.25$/hr
Memory Large	4.8 cores	38.4GB	0.44$/hr

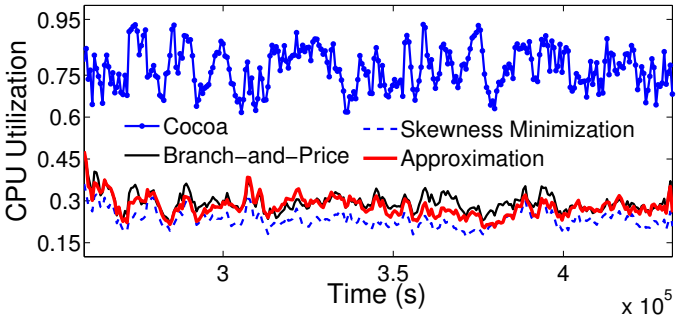

Fig. 6. A comparison of CPU utilization in *Cocoa*, VM consolidation strategies and pure static grouping strategies.

group buying deal type, we consider the combination of Google "High CPU" and "High memory" instances that offers the equivalent amount of resource as the group buying deal does. Then, we set the deal price as the price of the instance combination in Google Compute Engine.

In our simulation, we evaluate the performance of *Cocoa*, which combines the approximation static grouping strategy and prioritized learn-and-pack dynamic strategy. Static grouping is performed at the beginning of the service or whenever the number of jobs in the waiting queue reaches 20. In the prioritized learn-and-pack strategy, we set the size of deal classes as 5. We compare to the performance of a VM consolidation strategy (*i.e.*, Skewness Minimization [Xiao et al. 2013]), two pure static group buying organization strategies (*i.e.*, Branch-and-Price, Approximation), two pure dynamic grouping strategies (*i.e.*, Learn-and-Pack and Prioritized Learn-and- Pack), and a combined strategy (*i.e.*, Prioritized Learn-and-Pack + Branch-and-Price).

6.3. Resource Utilization

Comparing to VM Consolidation Strategies and Pure Static Strategies: We implement both Branch-and-Price and its polynomial-time approximation algorithm

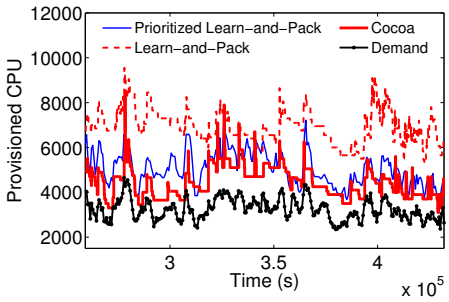

Fig. 7. A comparison of the CPU demand from all the jobs and the CPU provisioned in different group buying strategies over time.

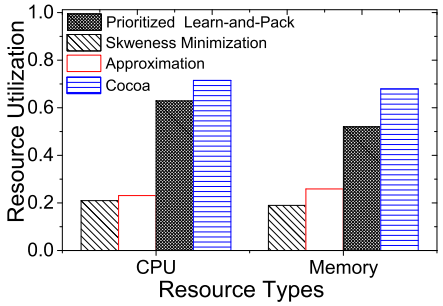

Fig. 8. The time averaged utilization of resource provided by all the group buying deals under different grouping strategies.

Table III. Provider Cost Comparison under Different Strategies

	Individual Instance Buying	Learn-and-Pack	Prioritized Learn-and-Pack	*Cocoa*	Prioritized Learn-and-Pack +Branch-and-Price
Cost ($)	1804	1682	1484	1248	1225
Cost Saving	-	6.7%	17.7%	30.8%	32.1%

as two pure static group buying strategies and compare their corresponding resource utilization. For comparison, we also evaluate the performance of a conventional VM consolidation strategy proposed in [Xiao et al. 2013], which tries to minimize *the skewness of resource usage*. For all these strategies, we perform static grouping for each batch of 20 jobs.

In Fig. 6, we plot the CPU utilization under above-mentioned strategies in a continuous period of 2 days. We can observe that *Cocoa* substantially outperforms all the other strategies, by smartly combining a static bin-packing strategy and an online knapsacking strategy (with prioritized classes) to handle resource holes due to job departures. In fact, the conventional VM consolidation strategy and static strategies achieve a utilization consistently lower than 50%.

Comparing to Pure Dynamic Strategies: We further investigate how pure dynamic group buying strategies can enhance resource utilization when serving small users. Fig. 7 shows the relationship of demand and supply in memory resource under different strategies, in the time period from the 4^{th} day to the 5^{th} day in the trace. We can observe that *Cocoa*, which uses the least resource to satisfy the demand of all the jobs, achieves the highest utilization among all the strategies. In addition, Learn-and-Pack, with no priority in dynamic group organization, shows inefficiency during scaling down when the total demand decreases. Compared to *Cocoa*, the pure dynamic Prioritized Learn-and-Pack strategy, without a waiting queue to buffer new jobs, shows more fluctuations in resource provisioning and lower utilization in demand peaks.

In Fig. 8, we show the time averaged resource utilization of all the group buying deals under different group buying strategies. Again, *Cocoa* outperforms pure strategies and the VM consolidation strategy. The Prioritized Learn-and-Pack strategy outperforms the Approximation and Skewness Minimization strategy, since it can better handle job dynamics by packing newly arrived jobs into resource "holes" in running group buying deals. *Cocoa* outperforms Prioritized Learn-and-Pack strategy alone with more balanced resource usage on CPU and memory. All the strategies have a higher utilization in CPU than in memory, this is because most of the jobs in the trace consume a larger amount of CPU resource than memory.

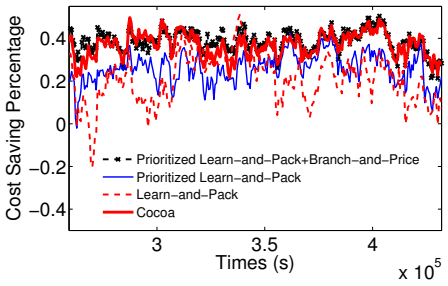

Fig. 9. Cost saving percentage for the cloud provider under different group buying strategies.

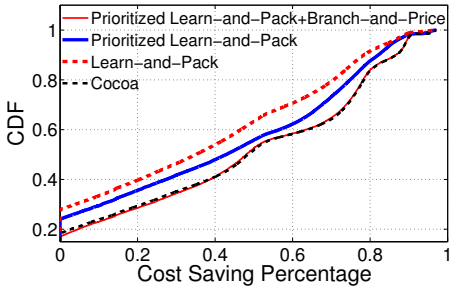

Fig. 10. Cumulative distribution function (CDF) for cost saving percentage of individual users (jobs) under different group buying strategies.

6.4. Cost Saving for the Provider

We analyze the cost saving for the provider by using *Cocoa*, instead of virtual machines, to serve cloud users. The performance results of pure static strategies and the Skewness Minimization VM consolidation strategy are not presented, since they are not efficient enough to save cost for the provider. However, we present the results from the Prioritized Learn-and-Pack + Branch-and-Price to compare the performance of *Cocoa* to the combined strategy with optimal static grouping algorithm.

In Table III, we present the cost of provider under different strategies, and calculate their cost saving percentage compared to normal individual instance purchasing. We can see that the strategies that combine both static and dynamic grouping algorithms outperform pure dynamic strategies by saving more than 30% of the cost. Considering static algorithms, *Cocoa* has a cost saving percentage similar to Prioritized Learn-and-Pack + Branch-and-Price strategy, which indicates that the approximation algorithm reaches a performance very close to the optimal yet impractical branch-and-price algorithm. In terms of dynamic algorithms, we can observe that by simply prioritize some deals over others, Prioritized Learn-and-Pack outperforms Learn-and-Pack by saving about 11% more cost.

Fig. 9 describes the dynamic fluctuation of cost saving percentages under different group buying strategies in a time period of two days. The combined strategies have a stable cost saving percentage of over 30%, while pure dynamic strategies have a more varied cost saving percentage. For some points in time, the cost saving even falls under 0. This is because when demands are low, there would be a large amount of idle resources in each group buying deal. In some extreme cases, resources are much under-utilized so that using individual instances to serve users are more cost-effective.

6.5. Cost Saving for Individual Cloud Users

To ensure cost efficiency to short jobs, we charge cloud users by minutes in *Cocoa*. Within each minute, a job i in group buying deal j only pays a portion of the price of the group buying deal it belongs to. Specifically, assume job i runs for a time period of t_i minutes with CPU and memory demand of CPU_i and $Memroy_i$, respectively. We calculate the *weight* of a job w_i as $\alpha CPU_i + \beta Memory_i$, where α and β are deemed as the "unit prices" of CPU and memory, respectively. Then we charge each job a fee that equals to $t_i \cdot w_i / \sum_i t_i \cdot w_i$ times the price of the entire group buying deal. The "unit prices" α and β are calculated by solving a linear equation according to the resource configuration and prices of two standard Google instances ("High CPU" and "High Memory"), such that for a job i, its weight w_i will equal to the cost of the combination of "High CPU" and "High Memory" instances that provide exactly the same amount of resources that job i has demanded. It is worth noting that in some under-

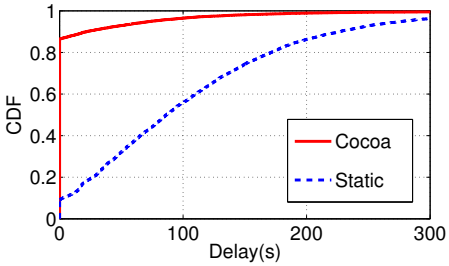

Fig. 11. Cumulative distribution function (CDF) for queuing delay in *Cocoa* and the pure static grouping.

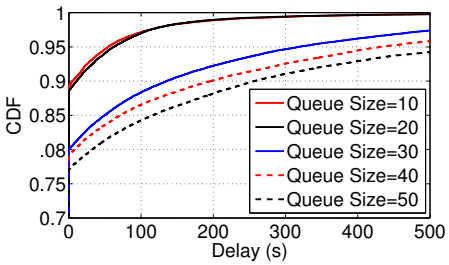

Fig. 12. Cumulative distribution function (CDF) for queuing delay in *Cocoa* with different waiting queue size.

utilized deals, the group buying fee of a user calculated above might even exceed its individual instance purchase fee, because the jobs might have also shared the cost of idle resources in a deal that they do not use. To ensure cost saving to users, we bound the a user's group buying fee by the price of the smallest individual instance that can satisfy her demand.

Fig. 10 plots the CDF of cost savings for individual users under different group buying strategies. With *Cocoa*, small jobs are consolidated for higher utilization. Due to resource sharing, each user has to pay less for unused resource, thus enjoying a price discount. We can observe that *Cocoa* and Prioritized Learn-and-Pack + Branch-and-Price, the latter being the optimal solution with a high complexity, have nearly identical performance, and outperform pure dynamic strategies, with nearly 40% of cloud users saving more than 60% of their cost. In pure dynamic strategies, Prioritized Learn-and-Pack consistently outperforms Learn-and-Pack strategy by a small margin.

6.6. Service Latency

We now investigate the service latency cloud jobs experience under different strategies. The service delay that a job experiences is the sum of the queuing delay and computation delay. Recall that a job might be put in a waiting queue to get packed into a group buying deal. We refer the time that a job stays in the waiting queue as its queuing delay. In addition, when there are multiple group buying deals to run jobs, we need to solve either the static packing problem or the dynamic packing problem to calculate the proper packing decision. We refer the time used to compute and solve the packing problems as the computation delay.

We first evaluate the queuing delay in *Cocoa*. Pure dynamic grouping strategies causes no queuing delay since they pack all jobs into group buying deals on their arrival. Therefore we only compare the queuing delay in *Cocoa* to pure static group buying strategies and present the results in Fig. 11. In the simulation, we run the static algorithm to pack jobs into group buying deals whenever there are 20 jobs in the waiting queue. Therefore, all the static strategies result in the same queuing delay. We can observe that static strategies result in a much larger queuing latency than *Cocoa*, with more than 40% of jobs experiencing a queuing delay longer than 100 seconds.

Next, we compare the computation delay under different strategies. Since the computation delay of the dynamic grouping strategy is much smaller than that of the static grouping strategies. We only compare *Cocoa* with the pure static Branch-and-Price strategy. For fair comparison, we rule out the computation delay of dynamic grouping in *Cocoa* and only compare the computation delay of the static strategy in *Cocoa* (*i.e.,* the approximation algorithm) with the Branch-and-Price strategy. We run both algorithms to pack jobs in our simulation, with each algorithm executed for 800 times. Both of the algorithms are implemented in the C programming language and are ex-

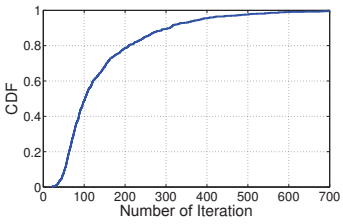

Fig. 13. Cumulative distribution function (CDF) for the number of master problem-subproblem iterations to achieve convergency on one node of the branch tree.

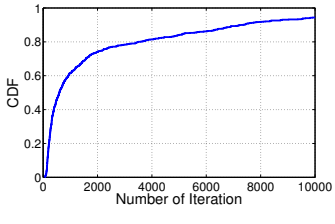

Fig. 14. Cumulative distribution function (CDF) for the total number of master problme-subproblem iterations to achieve convergency for the branch-and-price algorithm.

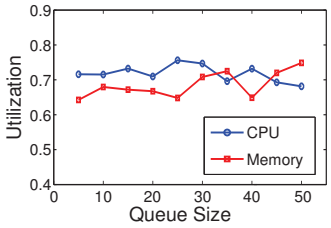

Fig. 15. A comparison of time averaged CPU and memory utilization in *Cocoa* with different waiting queue sizes.

ecuted on a server with an Intel Xeon E5620 16-core CPU, 16GB DDR3 memory and Red Hat 4.8.3-9 OS. The Branch-and-Price algorithm takes 13 minutes for each run on average. The approximation strategy finishes in 13 seconds in each run on average. In conclusion, compared to pure static strategies, *Cocoa* causes a smaller latency in both queuing and computation, thus reducing the total service delay.

6.7. Convergence Speed of the Static Grouping Strategy

Recall that the static grouping strategy adopts a branch-and-bound framework. On each node of the branch tree, the algorithm proceeds in an iterative way. Within each iteration, the allocation choosing problem and the allocation generation problem are solved to find feasible allocations that can potentially improve the result. We now evaluate the convergence speed of the static grouping algorithm, *i.e.,* the number of iterations it takes to get the result. In the simulation, we execute the static grouping strategy for 800 times and plot the average number of iterations ran on a branch node for each execution in Fig. 13. Nearly 50% of executions run less than 100 iterations in average for an branch node, and for 90% of the executions there are no more than 300 iterations on each node in average. The number of nodes on the branch tree is 36.7 in average of all the executions. We further illustrate the total number of iterations on all the nodes in one execution in Fig. 14. Although more than 60% of the executions converges with in 1000 iterations, around 5% of the executions need to run more than 10000 iterations to achieve convergency. Although the static grouping strategy can converge within hundreds of iterations for more than half of the cases, for a small portion of cases it is not efficient in terms of converge speed, which calls for the approximation algorithm that is more efficient in computation.

6.8. Impact of Parameters

Waiting queue Size: We now evaluate the impact of the size of waiting queue on the performance of *Cocoa*. We calculate the time averaged resource utilization of *Cocoa* under difference waiting queue sizes ranging from 5 to 50, and present the results in Fig. 15. We can observe that the resource utilization in *Cocoa* is generally not affected by the waiting queue size. For e both CPU and memory, their utilization maintain in a high level with fluctuations within 10%. In Fig. 12, we further investigate the impact of waiting queue size on queuing delay in the service. We can observe that larger queue size leads to higher latency since jobs need to wait a long time for the queue to be full to perform static grouping.

Deal Size: To understand how deal size impacts the cost saving of group buying, we conduct the group buying strategies in the settings where the amount of resources provided in a group buying deal is 1.5 and 2 times that specified in Table II, and present the results in Fig. 16. We observe a counter-intuitive phenomenon that while combined

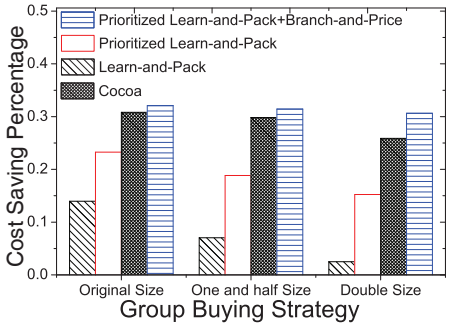

Fig. 16. Cost saving percentage compared to individual instance buying strategy with different group buying deal sizes.

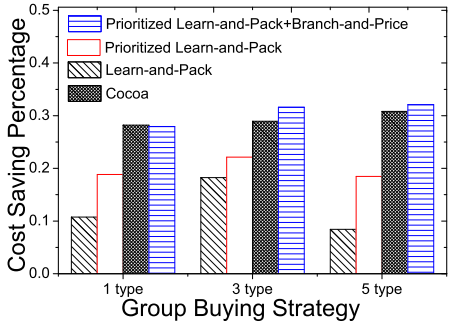

Fig. 17. Cost saving percentage compared to individual instance buying strategy with different number of group buying deal types.

strategies constantly outperform pure dynamic strategies, the performance of all the strategies decreases with the increase of the deal size. Generally, bigger deals can pack more jobs, thus generating more candidate consolidation choices and better consolidation efficiency. However, bigger deals also result in more resource wastage during the process of system-wide scaling down due to demand decrease, as resources are freed up. Compared to combined strategies, pure dynamic strategies, without carefully performing static group buying to better exploit bigger deals, suffer more from resource wastage with bigger deals.

Diversity of Deal Types: We now evaluate the impact of the diversity in group buying deals. Specifically, we compare the performance of a single deal type (Balanced) with that of 3 deal types (CPU small, Balanced, memory small), and show the result in Fig. 17. When there are more group buying deal types, the margin by which the combined strategies outperform pure dynamic strategies increases. This is because more types of group buying deals offer more choices for job consolidation in static group buying organization, which results in a higher utilization and lower cost. For pure dynamic group organization strategies, their decisions are made only according to the currently idle resources in each deal, which already has some variety. Thus, the diversity of group buying deal types would have little impact on dynamic strategies. Pure dynamic strategies get the best performance with 3 deal types, since the used deal types in this case in smaller compared to the other two cases, which leads to less resource wastage.

7. RELATED WORK

Group buying is a marketing strategy that offers consumers a lower price if they form groups and purchase in a collective way. To the best of our knowledge, the only works which apply the idea of group buying to computing systems were done by Stanojevic *et al*. [Stanojevic et al. 2011] and Lin *et al.* [Lin et al. 2013]. In [Stanojevic et al. 2011], a cooperative framework is proposed in which multiple ISPs jointly purchase IP transit in bulk to save individual cost. In [Lin et al. 2013], a three-stage auction framework is proposed to let secondary users with limited budgets collectively bid for high price spectrums. In this paper, we make the first attempt to exploit the group buying strategy in cloud market. We propose *Cocoa*, a novel group buying mechanism particularly adapted to the cloud computing context for small and short jobs, which brings benefits to both cloud users and the service provider.

The approach we use in *Cocoa* to organize users into group buying deals is most related to the literature of VM consolidation and job scheduling in the cloud context. In VM consolidation, VMs are packed into physical servers with the target of utilization improvement or energy saving. For example, Xu *et al.* [Xu et al. 2014a] study

the overhead of live migration for VM consolidation. Verma *et al.* [Verma et al. 2009] analyze an enterprise server workload trace and find significant potential of using consolidation for power saving in datacenters. Beloglazov *et al.* [Beloglazov and Buyya 2012] investigate the energy-performance trade-off of VM consolidation in the cloud and propose online deterministic algorithms as well as historical data based heuristics for single VM migration and dynamic VM consolidation problems. Xiao *et al.* [Xiao et al. 2013] introduce the concept of "skewness" as a metric of resource usage unevenness in a VM consolidation scheme, and propose a heuristic to minimize it while trying to save energy at the same time. Meng *et al.* [Meng et al. 2010] use a stochastic bin packing algorithm to consolidate VMs with dynamic bandwidth demands into the minimal number of physical servers while not violating the bandwidth capacity constraint. Beloglazov *et al.* [Beloglazov and Buyya 2013] study the side effect of VM consolidation as it may cause resource shortage and server overload, thus influencing the QoS of the hosted applications. Mishra *et al.* [Mishra et al. 2012] present heuristic algorithms which utilize live virtual machine migration techniques for not only server consolidation but also load balancing and hotspot mitigation. Jung *et al.* [Jung et al. 2010] propose a framework to adapt the VM consolidation scheme to the demand of multi-tier web applications while optimizing a global utility function of power consumption, performance and transient cost. Le *et al.* [Le et al. 2011] study the impact of VM placement policy on electricity cost in geographically distributed high performance computing clouds and design policies to place and migrate VMs across datacenters to take advantages of the differences in electricity prices and temperatures. In the case of job scheduling and resource allocation, Ghodsi *et al.* [Ghodsi et al. 2011] consider the problem of fair resource allocation in a system with multiple types of resources, and design an resource allocation strategy DRF that not only incentivizes users to share resources but also achieves strategy-proof, envy-free and Pareto efficient. Based on DRF, Bhattacharya *et al.* [Bhattacharya et al. 2013] design and implement H-DRF, an algorithm for multi-resource fair hierarchical scheduling, which can avoid the shortages of starvation and resource inefficiency in existing open source schedulers. Lee *et al.* [Lee et al. 2011] focus on the scheduling of data analytics applications and present a resource allocation and job scheduling framework that considers workload and hardware heterogeneity in addition to fairness. Yi *et al.* [Yi et al. 2016] study a strategy to exploit the scheduling flexibility of delay tolerant workloads and propose a pricing and resource allocation framework to meet the deadlines of the jobs.

Unlike dynamic VM consolidation, which deals with dynamics of long term workload in running VMs by live migration, grouping small and short jobs poses new challenges. To avoid the overhead of live migration when dealing with the strong dynamics of job arrivals and departures, we combine the static grouping strategy with a dynamic strategy in *Cocoa*. In terms of job scheduling and resource allocation, *Cocoa* is designed to benefit both the provider and users in the cloud context. Therefore, rather than fairness, *Cocoa* aims to achieve cost efficiency to cloud users and resource efficiency to the provider, by packing user jobs into the proper group buying deals.

8. CONCLUSION

In this paper, we present *Cocoa*, a container-based framework to implement group buying for cloud resources. In *Cocoa*, user jobs of complementary resource demands are grouped together and allocated to newly created group buying deals or existing deals with ideal resources with a price incentive, while the cloud provider can improve its resource utilization and save cost. Unlike prior VM-based workload consolidation, we adopt not only a static grouping strategy for batches of new jobs, but also a dynamic strategy to fill resource "holes" online as jobs finish and leave at different times. The static and dynamic grouping problems are modeled as a vector bin packing problem

and an online knapsack problem, respectively. Through extensive simulations based on a large amount of real-world cloud usage traces, we show that by smartly combining the static and online grouping strategies, *Cocoa* significantly outperforms existing workload consolidation strategies in terms of cost reduction, resource utilization, and service delay, and provides viable business opportunities for group buying in cloud computing.

REFERENCES

Amazon EC2 Spot Instance Pricing. 2014. http://aws.amazon.com/ec2/spot/pricing/. (2014).

Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. 1998. Branch-and-price: Column generation for solving huge integer programs. *Operations research* 46, 3 (1998), 316–329.

Anton Beloglazov and Rajkumar Buyya. 2012. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience* 24, 13 (2012), 1397–1420.

Anton Beloglazov and Rajkumar Buyya. 2013. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *Parallel and Distributed Systems, IEEE Transactions on* 24, 7 (2013), 1366–1379.

Arka A Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Hierarchical scheduling for diverse datacenter workloads. In *Socc' 13: Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 4.

J. Wilkes C. Reiss and J. Hellerstein. 2011. "Google Cluster-Usage Traces". http://code.google.com/p/googleclusterdata/. (2011).

Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1802–1813. DOI:http://dx.doi.org/10.14778/2367502.2367519

CloudSigma. 2014. http://www.cloudsigma.com/. (2014).

George B Dantzig and Philip Wolfe. 1960. Decomposition principle for linear programs. *Operations research* 8, 1 (1960), 101–111.

Nikhil R. Devanur and Thomas P. Hayes. 2009. The Adwords Problem: Online Keyword Matching with Budgeted Bidders Under Random Permutations. In *EC'09: Proceedings of the 10th ACM Conference on Electronic Commerce*. ACM, New York, NY, USA, 71–78. DOI:http://dx.doi.org/10.1145/1566374.1566384

Docker. 2014. Docker install docs. https://docs.docker.com/installation/#installation. (2014).

Khaled Elmeleegy. 2013. Piranha: Optimizing Short Jobs in Hadoop. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 985–996. DOI:http://dx.doi.org/10.14778/2536222.2536225

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 323–336. http://dl.acm.org/citation.cfm?id=1972457.1972490

Gagan Goel and Aranyak Mehta. 2008. Online budgeted matching in random input models with applications to adwords. In *SODA'08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 982–991. DOI:http://dx.doi.org/citation.cfm?id=1347082.1347189

Chien-Ju Ho and Jennifer Wortman Vaughan. 2012. Online Task Assignment in Crowdsourcing Markets.. In *AAAI*, Vol. 12. 45–51.

Gueyoung Jung, Matti A Hiltunen, Kaustubh R Joshi, Richard D Schlichting, and Calton Pu. 2010. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *ICDCS'10: Proceedings of Thirtieth IEEE International Conference on Distributed Computing Systems*. IEEE, Genova, 62–73. DOI:http://dx.doi.org/10.1109/ICDCS.2010.88

Kien Le, Ricardo Bianchini, Jingru Zhang, Yogesh Jaluria, Jiandong Meng, and Thu D. Nguyen. 2011. Reducing Electricity Cost Through Virtual Machine Placement in High Performance Computing Clouds. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, New York, NY, USA, Article 22, 12 pages. DOI:http://dx.doi.org/10.1145/2063384.2063413

Gunho Lee, Byung-Gon Chun, and H. Katz. 2011. Heterogeneity-aware Resource Allocation and Scheduling in the Cloud. In *HotCloud'11: Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud*

Computing. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2170444.2170448

Peng Lin, Xiaojun Feng, Qian Zhang, and Mounir Hamdi. 2013. Groupon in the air: A three-stage auction framework for spectrum group-buying. In *Proc. of INFOCOM*. IEEE, Turin, 2013–2021. DOI:http://dx.doi.org/10.1109/INFCOM.2013.6567002

Mohammad Mahdian and Qiqi Yan. 2011. Online bipartite matching with random arrivals: an approach based on strongly factor-revealing lps. In *Proc. of SODA*. ACM, New York, NY, USA, 597–606. DOI:http://dx.doi.org/10.1145/1993636.1993716

Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. 2010. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *In Proc. of INFOCOM*. IEEE, San Diego, CA, 1–9. DOI:http://dx.doi.org/10.1109/INFCOM.2010.5461930

Mayank Mishra, Anwesha Das, Purushottam Kulkarni, and Anirudha Sahoo. 2012. Dynamic resource management using virtual machine migrations. *Communications Magazine, IEEE* 50, 9 (2012), 34–40.

Timothy Prickett Morgan. 2014. Google Runs All Software In Containers. http://www.enterprisetech.com/2014/05/28/google-runs-software-containers/. (2014).

OpenVz. 2014. http://openvz.org/Main_Page. (2014).

Peter Orlik and Hiroaki Terao. 1992. Arrangements of hyperplanes, volume 300 of Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]. (1992).

Boaz Patt-Shamir and Dror Rawitz. 2012. Vector bin packing with multiple-choice. *Discrete Applied Mathematics* 160, 10 (2012), 1591–1600.

Jerome Petazzoni. 2013. Lightweight Virtualization with Linux Containers and Docker. http://tech.yandex.com/events/yac/2013/talks/14/. (2013).

Serge A Plotkin, David B Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research* 20, 2 (1995), 257–301.

David M Ryan and Brian A Foster. 1981. An integer programming approach to scheduling. *Computer scheduling of public transport urban passenger vehicle and crew scheduling* (1981), 269–280.

Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proc. of EuroSys*. ACM, New York, NY, USA, 275–287. DOI:http://dx.doi.org/10.1145/1272996.1273025

Rade Stanojevic, Ignacio Castro, and Sergey Gorinsky. 2011. CIPT: using tuangou to reduce IP transit costs. In *Proc. of CoNEXT*. ACM, New York, NY, USA, 17:1–17:12. DOI:http://dx.doi.org/10.1145/2079296.2079313

David Strauss. 2013. Containers-Not Virtual Machines-Are the Future Cloud. http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud. (2013).

AadW. van der Vaart and JonA. Wellner. 1996. Weak Convergence. In *Weak Convergence and Empirical Processes*. Springer New York, 16–28. DOI:http://dx.doi.org/10.1007/978-1-4757-2545-2_3

Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. 2009. Server Workload Analysis for Power Minimization Using Consolidation. In *Proc. of ATC*. USENIX Association, Berkeley, CA, USA, 1.

virtuozzo. 2014. http://www.parallels.com/virtuozzo. (2014).

Wei Wang, Di Niu, Baochun Li, and Ben Liang. 2013. Dynamic cloud resource reservation via cloud brokerage. In *ICDCS'13: Proceedings of Thirty-third IEEE International Conference on Distributed Computing Systems*. IEEE, Philadelphia, PA, 400–409. DOI:http://dx.doi.org/10.1109/ICDCS.2013.20

Zhen Xiao, Weijia Song, and Qi Chen. 2013. Dynamic resource allocation using virtual machines for cloud computing environment. *Parallel and Distributed Systems, IEEE Transactions on* 24, 6 (2013), 1107–1117.

Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. 2014a. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proc. IEEE* 102, 1 (2014), 11–31.

Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. 2014b. iaware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Trans. Comput.* 63, 12 (2014), 3012–3025.

Xiaomeng Yi, Fangming Liu, Zongpeng Li, and Hai Jin. 2016. Flexible Instance: Meeting Deadlines of Delay Tolerant Jobs in The Cloud with Dynamic Pricing. In *Proc. of ICDCS*.

Xiaomeng Yi, Fangming Liu, Jiangchuan Liu, and Hai Jin. 2014. Building a network highway for big data: architecture and challenges. *IEEE Network* 28, 4 (2014), 5–13.

Received February 2007; revised March 2009; accepted June 2009