Automatic Scaling of Internet Applications for Cloud Computing Services

Zhen Xiao, *Senior Member, IEEE,* Qi Chen, and Haipeng Luo

Abstract—Many Internet applications can benefit from an automatic scaling property where their resource usage can be scaled up and down automatically by the cloud service provider. We present a system that provides automatic scaling for Internet applications in the cloud environment. We encapsulate each application instance inside a virtual machine (VM) and use virtualization technology to provide fault isolation. We model it as the Class Constrained Bin Packing (CCBP) problem where each server is a bin and each class represents an application. The class constraint reflects the practical limit on the number of applications a server can run simultaneously. We develop an efficient semi-online color set algorithm that achieves good demand satisfaction ratio and saves energy by reducing the number of servers used when the load is low. Experiment results demonstrate that our system can improve the throughput by 180% over an open source implementation of Amazon EC2 and restore the normal QoS five times as fast during flash crowds. Large scale simulations demonstrate that our algorithm is extremely scalable: the decision time remains under 4 seconds for a system with 10,000 servers and 10,000 applications. This is an order of magnitude improvement over traditional application placement algorithms in enterprise environments.

Index Terms—cloud computing, virtualization, auto scaling, CCBP, green computing.

1 Introduction

One of the often cited benefits of cloud computing service is the resource elasticity: a business customer can scale up and down its resource usage as needed without upfront capital investment or long term commitment. The Amazon EC2 service [1], for example, allows users to buy as many virtual machine (VM) instances as they want and operate them much like physical hardware. However, the users still need to decide how much resources are necessary and for how long. We believe many Internet applications can benefit from an auto scaling property where their resource usage can be scaled up and down automatically by the cloud service provider. A user only needs to upload the application onto a single server in the cloud, and the cloud service will replicate the application onto more or fewer servers as its demand comes and goes. The users are charged only for what they actually use – the so-called "pay as you go" model.

Figure 1 shows the typical architecture of data center servers for Internet applications. It consists of a load balancing switch, a set of application servers, and a set of backend storage servers. The front end switch is typically a Layer 7 switch [2] which parses application level information in Web requests and forwards them to the servers with the corresponding applications running. The switch sometimes runs in a redundant pair for fault tolerance. Each application can run on multiple server machines and the set of their running instances are often managed by some clustering software such as WebLogic [3]. Each server machine can host multiple applications. The applications store their state information in the backend storage servers. It is important that the applications themselves are stateless so that they can be replicated safely. The storage servers may also become overloaded, but the focus of this work is on the application tier. The Google AppEngine service, for example, requires that the applications be structured in such a two tier architecture and uses the BigTable as its scalable storage solution [4]. A detailed comparison with AppEngine is deferred to Section 7 so that sufficient

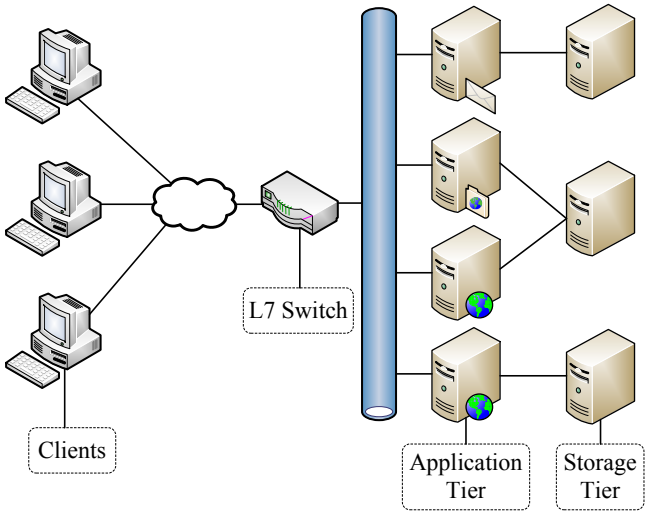


Fig. 1. Two-tiered architecture for Internet applications

• *Z. Xiao and Q. Chen are with the Department of Computer Science at Peking University, Beijing, China. Email: {xiaozhen, chenqi}@net.pku.edu.cn. H. Luo is with the Department of Computer Science at Princeton University, Princeton, New Jersey. Email: haipengl@cs.princeton.edu. The contact author is Zhen Xiao.*

background can be established. Some distributed data processing applications cannot be mapped into such a tiered architecture easily and thus are not the target of this work. We believe our architecture is representative of a large set of Internet services hosted in the cloud computing environment.

Even though the cloud computing model is sometimes advocated as providing infinite capacity on demand, the capacity of data centers in the real world is finite. The illusion of infinite capacity in the cloud is provided through statistical multiplexing. When a large number of applications experience their peak demand around the same time, the available resources in the cloud can become constrained and some of the demand may not be satisfied. We define the demand satisfaction ratio as the percentage of application demand that is satisfied successfully. The amount of computing capacity available to an application is limited by the placement of its running instances on the servers. The more instances an application has and the more powerful the underlying servers are, the higher the potential capacity for satisfying the application demand. On the other hand, when the demand of the applications is low, it is important to conserve energy by reducing the number of servers used. Various studies have found that the cost of electricity is a major portion of the operation cost of large data centers. At the same time, the average server utilization in many Internet data centers is very low: real world estimates range from 5% to 20% [5] [6]. Moreover, work [7] has found that the most effective way to conserve energy is to turn the whole server off. The application placement problem is essential to achieving a high demand satisfaction ratio without wasting energy.

In this paper, we present a system that provides automatic scaling for Internet applications in the cloud environment. Our contributions include the following:

- We summarize the automatic scaling problem in the cloud environment, and model it as a modified Class Constrained Bin Packing (CCBP) problem where each server is a bin and each class represents an application. We develop an innovative auto scaling algorithm to solve the problem and present a rigorous analysis on the quality of it with provable bounds. Compared to the existing Bin Packing solutions, we creatively support item departure which can effectively avoid the frequent placement changes [1] caused by repacking.
- We support green computing by adjusting the placement of application instances adaptively and putting idle machines into the standby mode. Experiments and simulations show that our algorithm is highly efficient and scalable which can achieve high demand satisfaction ratio, low placement change frequency, short request response time, and good energy saving.

1. starting an application instance at a new server or stopping an application instance at an existing server

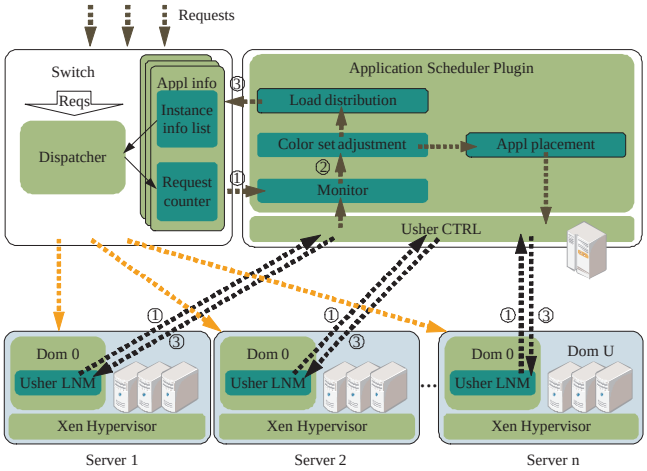


Fig. 2. Overview of the System Architecture

- We build a real cloud computing system which supports our auto scaling algorithm. We compare the performance of our system with an open source implementation of the Amazon EC2 auto scaling system in a testbed of 30 Dell PowerEdge blade servers. Experiments show that our system can restore the normal QoS five times as fast when a flash crowd happens.
- We use a fast restart technique based on virtual machine (VM) suspend and resume that reduces the application start up time dramatically for Internet services.

The rest of the paper is organized as follows. Section 2 presents the architecture of the system. Section 3 formulates the auto scaling problem. Section 4 describes the details of our algorithm. Experiment and simulation results are presented in Sections 5 and 6, respectively. Section 7 describes the related work and Section 8 concludes.

2 System Architecture

The architecture of our system is shown in figure 2. We encapsulate each application instance inside a virtual machine (VM). The use of VMs is necessary to provide isolation among untrusted users. Both Amazon EC2 and Microsoft Azure use VMs in their cloud computing offering. Each server in the system runs the Xen hypervisor which supports a privileged domain 0 and one or more domain U [8]. Each domain U encapsulates an application instance, which is connected to a shared network storage (i.e., the storage tier). The multiplexing of VMs to PMs (Physical Machines) is managed using the Usher framework [9]. (We use the terms "server", "PM", and "node" interchangeably in this paper.) The main logic of our system is implemented as a set of plug-ins to Usher. Each node runs a Usher local node

manager (LNM) on domain 0 which keeps track of the set of applications running on that node and the resource usage of each application. A L7 switch is in charge of forwarding requests and responses.

The schedule procedure of our system can be described as follows:

- The LNM at each node and the L7 switch collect the application placement, the resource usage of each instance, and the total request number of each application periodically. Then the information is forwarded to the Usher central controller (Usher CTRL) where our "Application Scheduler" runs.
- The Application Scheduler is invoked periodically to make the following decisions:
  - application placement: for each application, we need to decide the set of servers its instances run on.
  - load distribution: for each application, we need to predict its future resource demands based on the request rate and past statistics, and then decide how to allocate its load among the set of running instances. The load of an Internet application is largely driven by the rate of user requests. We profile the application to estimate the average load incurred by each request. Then we analyze the queues of pending requests in L7 switch to predict the load on the servers.
- The decisions are forwarded to the LNM and the L7 switch for execution. The list of action items for each node includes:
  - standby or wake up instructions
  - application starts and stops
  - the allocation of local resource among the applications

  The LNM at the node adjusts the local resource allocation of the VMs encapsulating the applications. Xen can change the CPU allocation among the VMs by adjusting their weights in its CPU scheduler. Memory allocation among the VMs can be adjusted using the ballooning technique. After that the Scheduler notifies the L7 switch of the new configuration including:
  - the list of applications
  - for each application, the location of its running instances and the probability of request distribution among them

  The L7 switch then starts processing Web requests according to the new configuration.

The decision interval of the Scheduler depends on how responsive we want to be to application demand change. Frequent placement changes are disruptive to application performance and should be avoided.

It may seem from the discussion above that the Usher CTRL is a central point of failure. Fortunately, it is not involved in the normal processing paths of user requests. Its failure only disrupts the updates on the load distribution and application placement policy. Incoming user requests can still be processed by the existing application instances. Should a higher degree of fault-tolerance be desired, we can use the hot mirroring technology developed by our group to create a backup copy of the Usher CTRL [10].

We also notice that complicated applications can take a long time (several minutes or much longer) to start and finish all the initializations. Since we run each application inside a VM, its entire running state can be suspended to the disk and then resumed at a later time. We take advantage of this feature to bypass the application start process by suspending a fully started and initialized application instance to the disk. It can then be resumed to start the application instance at the desired server. The resumption time is independent of the start up time of the application. It depends mostly on how fast the server can read the suspended VM file from the disk. With modern disk technology, the resumption time is typically quite short, which reduces the cost of placement changes significantly.

We evaluate the effectiveness of this technique using a RUBiS [11] service within JBoss on Dell PowerEdge blade servers with Intel 5620 CPU and 10K RPM SAS disks. For this relatively simple application, the start up time is close to 100 seconds, independent of the memory size. We consider the application as fully started when it can respond to a user request successfully. By resuming a previously created VM snapshot, we can reduce the start time significantly for small sized VMs: the start up time is reduced by 70% for a VM with 1G memory.

To support green computing, we put idle servers into the standby mode so that they can be waken up quickly in an on-demand manner. We measured the electric power consumption under various TPC-W workloads with the built-in watt-meter in our blade systems. We find that an idle blade server consumes about 130 Watts, a fully utilized server consumes about 205 Watts, while a server in standby mode only consumes about 20 Watts. Therefore, putting an idle server into standby mode can save about 85% energy, which inspires us to hibernate the idle servers when the load of the whole system is low, especially at night. When the demand increases, We can easily resume a host from standby mode to active state by using Wake-on-LAN (WOL) technology. According to Linux documentation [12], the standby-to-active transition time is 1-2 seconds for ACPI S1 state (standby) and 3-5 seconds for ACPI S3 state (suspend to ram). This is sufficient fast to react to changes in resource demands.

3 Problem Definition

In this section, we will summarize the automatic scaling problem into formulation.

For an Internet application, there exist multi-dimensional resource demands, such as CPU, memory, network bandwidth, disk I/O bandwidth etc. Among them, we choose CPU and memory as the representative

resources to be considered. This is modeled after Google AppEngine which hosts mostly e-commerce applications and charges the users based on their CPU consumption [4]. For those applications, memory is typically the determining factor on how many applications a server can run simultaneously, while CPU is the target resource we need to allocate among the application instances. However, we can deal with other types of bottleneck resources as well (e.g, replace memory with disk space for streaming media applications). We cannot handle applications with a larger number of *simultaneous* bottleneck resources (CPU, memory, disk, network I/O, etc.), nor can any of the existing commercial cloud computing offering including Google AppEngine and Amazon EC2 auto-scaling systems. This is typically not a problem in practice. A variant of the bin packing problem called "vector bin packing" seemingly considers multi-dimensional constraints, but cannot be applied in this context as will be explained in Section 7.

The auto scaling problem we want to solve is defined as follows: Suppose we have a server set S on which we need to run a set of applications (denoted as A). The CPU capacity of server s ($s \in S$) is C_s, the maximum number of application instances which can run on server s simultaneously according to memory factor is M_s, and the CPU demand of application a ($a \in A$) is c_a. Let P to be the application placement matrix ($P_{a,s} = 1$ means that application a has an instance running on server s, otherwise $P_{a,s} = 0$) and L to be the application load distribution matrix ($L_{a,s}$ is the CPU resource allocated on server s for application a). E is the energy consumption of an active server during a decision interval. Then as mentioned before, the inputs to our Application Scheduler contains the current application placement matrix P^*, the predicted CPU demand of each application (c_a) and the CPU and memory resource capacity of each server (C_s and M_s). The outputs contain new application placement matrix P and load distribution matrix L.

Our goals are to maximize the demand satisfaction ratio, minimize the placement change frequency and minimize energy consumption. Our optimization objectives can be expressed as follows:

$$(i) \quad \text{Maximize} \quad \sum_{a \in A} \sum_{s \in S} L_{a,s} \tag{1}$$

$$(ii) \quad \text{Minimize} \quad \sum_{a \in A} \sum_{s \in S} |P_{a,s} - P^*_{a,s}| \tag{2}$$

$$(iii) \quad \text{Minimize} \quad E * \left| \{s \mid \sum_{a \in A} P_{a,s} > 0\} \right| \tag{3}$$

with the constraints:

$$L_{a,s} \geq 0 \qquad \forall a \in A, \forall s \in S \tag{4}$$

$$P_{a,s} = 0 \Rightarrow L_{a,s} = 0 \qquad \forall a \in A, \forall s \in S \tag{5}$$

$$\sum_{a \in A} I_{a,s} \leq M_s \qquad \forall s \in S \tag{6}$$

$$\sum_{a \in A} L_{a,s} \leq C_s \qquad \forall s \in S \tag{7}$$

$$\sum_{s \in S} L_{a,s} \leq c_a \qquad \forall a \in A \tag{8}$$

To simplify the problem described above, we make the assumption that the servers are homogeneous with uniform capacity (C_s and M_s). Then the auto scaling problem is similar to the Class Constrained Bin Packing (CCBP) problem when we label each application as a class and treat the CPU demands of all $|A|$ classes as the items which need to be packed into $|S|$ bins. The only difference is that the CCBP problem does not have the "Minimize the placement change frequency" goal. Therefore, in order to solve our problem, we modified the CCBP model to support the "Minimize the placement change frequency" goal and provide a new enhanced semi-online approximation algorithm to solve it in the next section. Note that the equations above are just a formal presentation of the goals and constraints of our problem. We do not need to solve them directly.

4 CCBP for Automatic Scaling of Resources

In the traditional bin packing problem, a series of items of different sizes need to be packed into a minimum number of bins. The class constrained version of this problem divides the items into classes or colors. Each bin has capacity v and can accommodate items from at most c distinct classes. It is "class constrained" because the class diversity of items packed into the same bin is constrained. The goal is to pack the items into a minimum number of bins.

We can model our resource allocation as the Class Constrained Bin Packing (CCBP) problem where each server is a bin and each class represents an application. Items from a specific class represent the resource demands of the corresponding application. The class constraint reflects the practical limit on the number of applications a server can run simultaneously. For J2EE applications, for example, memory is typically the bottleneck resource. The capacity of a bin represents the amount of resources available at a server for all its applications. We assume that the servers are homogeneous with uniform capacity. This assumption will be relaxed later in the section. In practice, the number of servers available is finite. Our algorithm handles the case when all bins are used up. The size of an item represents an amount of load for the corresponding application. By making all items the same unit size, we can represent the item size as a unit of load equal to a specific fraction of the server capacity. This is called the "load unit". The capacity v of a bin thus represents how many units of load a server can accommodate. The number of items waiting to be packed from a specific class represents the amount of resource needed by the corresponding application. The resource needs of applications can vary with time. This is modeled as item arrivals and departures: load increases correspond to arrivals of new items, while load decreases correspond to departure of already packed items.

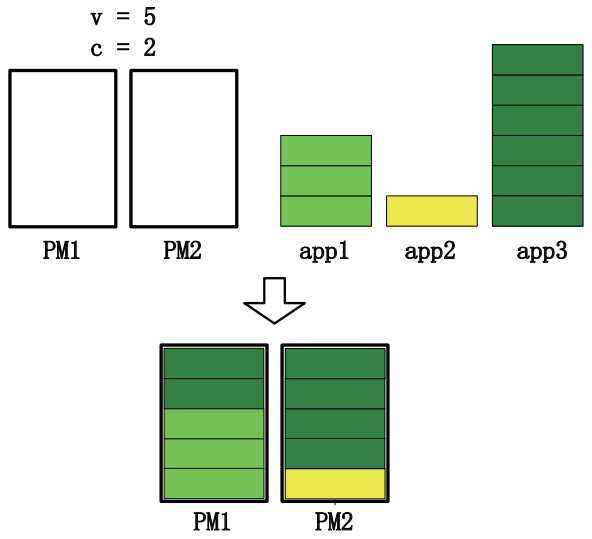


Fig. 3. Class Constrained Bin Packing (CCBP)

Most online CCBP algorithms do not support item departure. Work [13] has a general discussion on item departure, but does not describe any specific algorithm. Instead they focus on proving the bounds on the approximation ratio for any removal algorithm. Their result does not apply here because they do not repack items after the departure. A key contribution of our algorithm is the support for item departure which is essential to maintaining good performance in a cloud computing environment where the resource demands of Internet applications can vary dynamically.

Figure 3 shows an example where we have two servers and three applications. Each server can run at most two applications simultaneously (i.e., c=2). We set the load unit to 20% (i.e., a unit of load represented by an item equals 20% the capacity of a server). Each server can thus satisfy 5 units of load (i.e., v=5). Let the resource demands of the applications be 60%, 20%, and 120% of the server capacity, respectively. This translates into 3, 1, and 6 items to be packed for their corresponding classes.

The CCBP problem (like its traditional, non-class constrained version) is NP-hard [13]. A number of offline approximation algorithms have been developed in the literature [14] [15]. These algorithms assume that the entire input sequence of items (including their departure) is known in advance. This is unrealistic in our environment since application resource demands can change unexpectedly. In addition, they make no effort to minimize the shuffling of already packed items, which can lead to significant overhead due to application placement changes.

A key observation of our work is: not all item movements are equally expensive. Recall that we need to make two decisions in our system: application placement and load distribution. Creating a new application instance on a server is expensive. In the CCBP model, this corresponds to packing the first item of a class into a bin. Adjusting the load distribution, on the other hand, is a much cheaper operation. It involves updating the L7 switch with the new distribution policy. In the CCBP model, this corresponds to adjusting the number of items of an existing class in a bin. Based on this observation, we develop a semi-online algorithm for CCBP which packs the current item without any knowledge of subsequent items in the input sequence. Our key idea is to minimize application placement changes by adjusting the existing load distribution. In the following, we will describe the details of our algorithm.

4.1 Details of our algorithm

Our algorithm belongs to the family of color set algorithms [13], but with significant modification to adapt to our problem. A detailed comparison with the existing algorithm is deferred to Section 7 so that sufficient background can be established. We label each class of items with a color and organize them into color sets as they arrive in the input sequence. The number of distinct colors in a color set is at most c (i.e., the maximum number of distinct classes in a bin). This ensures that items in a color set can always be packed into the same bin without violating the class constraint. The packing is still subject to the capacity constraint of the bin. All color sets contain exactly c colors except the last one which may contain fewer colors.

Items from different color sets are packed independently. A greedy algorithm is used to pack items within each color set: the items are packed into the current bin until the capacity is reached. Then the next bin is opened for packing. Thus each color set has at most one unfilled (i.e., non-full) bin. Note that a full bin may contain fewer than c colors. When a new item from a specific color set arrives, it is packed into the corresponding unfilled bin. If all bins of that color set are full, then a new bin is opened to accommodate the item.

4.1.1 *Application load increase*

The load increase of an application is modeled as the arrival of items with the corresponding color. A naive algorithm is to always pack the item into the unfilled bin if there is one. If the unfilled bin does not contain that color already, then a new color is added into the bin. This corresponds to the start of a new application instance which is an expensive operation. Instead, our algorithm attempts to make room for the new item in a currently full bin by shifting some of its items into the unfilled bin. Let c_1 be the color of the new item and c_2 be any of the existing colors in the unfilled bin. We search for a bin which contains items of both colors. Let b be such a bin. Then we move an item of color c_2 from bin b to the unfilled bin. This makes room for an item in bin b where we pack the new item.

If we cannot find a bin which contains both colors, we see if we can shift the items using a third color c_3 as the intermediate. More specifically, we search for two bins:

- bin b_1 contains colors c_1 and c_3
- bin b_2 contains colors c_2 and c_3

If we can find such two bins, we proceed as follows:

- move an item of color c_2 from bin b_2 to the unfilled bin
- move an item of color c_3 from bin b_1 to bin b_2
- pack the item in bin b_1

This process is illustrated in Figure 4 (left). (Recall that v is the capacity of the bin and c is the class constraint.) More generally, we can have a chain of colors c_1, ..., c_k such that

- c_1 is the color of the new item
- c_k is an existing color in the unfilled bin
- every two adjacent colors in the chain share a bin[2]

The length of the chain is bounded by the number of colors in the color set (i.e., the class constraint) . As long as such a chain exists, we can accommodate the new item by shifting the existing items along the chain. Note that the item movements here are hypothetical and used only to calculate the new load distribution. No physical movement of any application occurs. Also note that the chain length is bounded by a constant and does not increase with the numbers of applications or servers in the system. If we cannot find such a chain, the new color has to be added into the unfilled bin which requires starting a new application instance. If the color set has no unfilled bin, then a new bin is allocated. If all bins are used up, then the load increase cannot be satisfied.

4.1.2 Application load decrease

The load decrease of an application is modeled as the departure of previously packed items. Note that the departure event here is associated with a specific color, not with a specific item. The algorithm has the freedom to choose which item of that color to remove.

The challenge here is to maintain the property that each color set has at most one unfilled bin. Our departure algorithm works as follows. If the color set does not have an unfilled bin, we can remove any item of that color and the resulting bin becomes the unfilled bin. Otherwise, if the unfilled bin contains the departing color, a corresponding item there can be removed directly. In all other cases, we need to remove an item from a currently full bin and then fill the hole with an item moved in from somewhere else. Let c_1 be the departing color and c_2 be any of the colors in the unfilled bin. We need to find a bin which contains items of both colors. Let b be such a bin. We remove the departing item from bin b and then move in an item of color c_2 from the unfilled bin. More generally, we can find a chain of colors and fill the hole of the departing item by shifting the existing items along the chain. The procedure is similar to the previous case for application load increase. Figure 4 (right) illustrates this process for a chain with three colors.

If we cannot find such a chain, we start a new application instance to fill the hole:

- remove an item of the departing color from any bin which contains that color.
- select a color c_2 in the unfilled bin and add that color into the departing bin.
- move an item of color c_2 from the unfilled bin to the departing bin.

If the unfilled bin becomes empty, we can remove it from the color set and shut down the corresponding server since all application instances there receive no load.

It might seem counter-intuitive that a decrease in application load can result in the start of a new application instance. However, this is inevitable if we want to consolidate servers to save energy.

4.1.3 Application joins and leaves

When the last item of a specific color leaves, that color can be removed from its color set. This corresponds to shutting down the last instance of an application when its load reduces to zero. [3] When a color is removed from its color set, that color set becomes unfilled (i.e., non-full). The challenge here is to maintain the property that there is at most one unfilled color set in the system. This is important because each color set is packed independently.

Our basic idea is to fill up the unfilled sets (except the last one) while minimizing its impact on the existing color assignment. We first check if there are any pending requests to add new colors into the system. If there are, we allocate the new colors to the unfilled sets first using the following **add_new_colors** procedure.

procedure **add_new_colors**:

- Sort the list of unfilled color sets in descending order of their cardinality.
- Use a greedy algorithm to add the new colors into those sets according to their positions in the list.
- If we run out of the new colors before filling up all but the last unfilled sets, use the **consolidate_unfilled_sets** procedure below to consolidate the remaining unfilled sets until there is only one left.
- If there are still new colors left after filling up all unfilled sets in the system, we partition the remaining new colors into additional color sets using a greedy algorithm.

The **consolidate_unfilled_sets** procedure below consolidates unfilled sets in the system until there is only one left.

procedure **consolidate_unfilled_sets**:

- Sort the list of unfilled color sets in descending order of their cardinality
- Use the last set in the list (with the fewest colors) to fill the first set in the list (with the most colors)

2. Note that the "share a bin" relation between the colors is not transitive.

3. Another possibility is to leave the last instance there to make sure that each application has at least one instance running in the system.

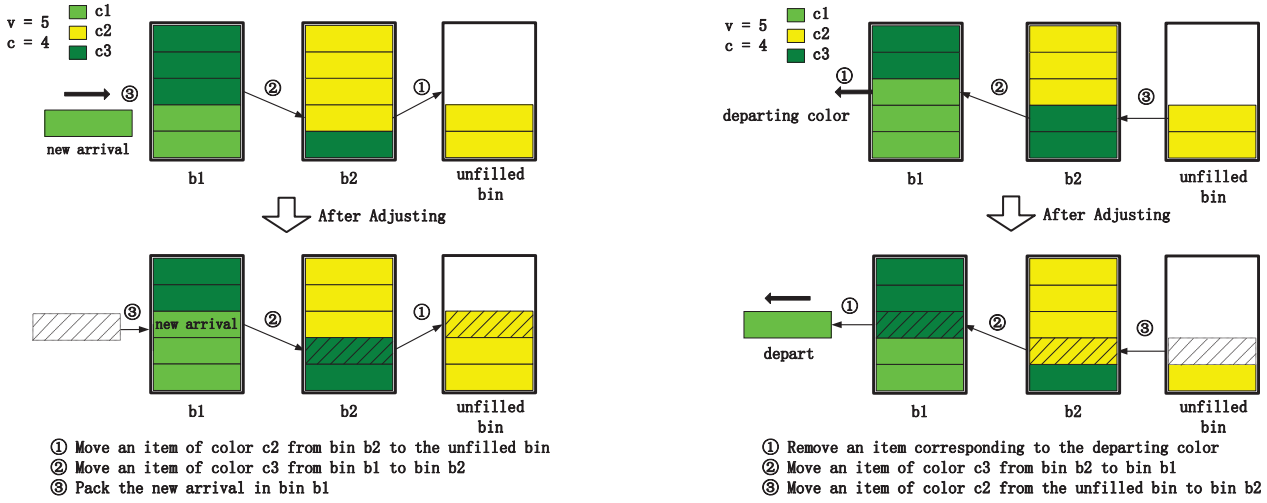


Fig. 4. arrival of a new item (left) and departure of an existing item (right)

through the **fill** procedure below. Remove the resulting full set or empty set from the list.
- Repeat the previous step until there is only one unfilled set left in the list.

The **fill(s_1, s_2)** procedure below uses the colors in set s_1 to fill the set s_2.

procedure **fill(s_1, s_2)**:
- Sort the list of colors in s_1 in ascending order of their numbers of items.
- Add the first color in the list (with the fewest items) into s_2. Use "item departure" operation in s_1 and "item arrival" operation in s_2 to move all items of that color from s_1 to s_2. Then remove that color from the list.
- Repeat the above step until either s_1 becomes empty or s_2 becomes full.

4.2 Analysis of the approximation ratio

The quality of a polynomial time algorithm A is measured by its approximation ratio $R(A)$ to the optimal algorithm OPT:

$$R(A) = \lim_{n\to\infty} \sup_{OPT(\sigma)=n} \frac{A(\sigma)}{OPT(\sigma)}$$

where σ is the list of the input sequence and $A(\sigma)$ and $OPT(\sigma)$ are the number of bins used under the A algorithm and the optimal algorithm, respectively [16]. Since the problem is NP-hard, no polynomial time optimal algorithm exists. Hence, the OPT algorithm here serves as an unrealistic upper bound.

To analyze the approximation ratio of our algorithm, we define the following notation:
- m: the total number of colors in the system
- l_i: the number of items of the i^{th} color
- c: the class constraint
- v: the capacity of a bin

Let $k = \lceil m/c \rceil$ and the average load $L = \sum_{i=1}^{m} l_i/m$. We have

$$OPT(\sigma) \geq \lceil \frac{\sum_{i=1}^{m} l_i}{v} \rceil = \lceil \frac{m \cdot L}{v} \rceil$$

$$\begin{aligned} A(\sigma) &= \sum_{i=1}^{k-1} \lceil \frac{\sum_{j=(i-1)\cdot c+1}^{i\cdot c} l_j}{v} \rceil + \lceil \frac{\sum_{i=(k-1)\cdot c+1}^{m} l_i}{v} \rceil \\ &\leq \lceil (\sum_{i=1}^{k-1} \frac{\sum_{j=(i-1)\cdot c+1}^{i\cdot c} l_j}{v}) + \frac{\sum_{i=(k-1)\cdot c+1}^{m} l_i}{v} \rceil \\ &\quad +(k - \lceil \frac{k}{v} \rceil) \\ &= \lceil \frac{m \cdot L}{v} \rceil + (k - \lceil \frac{k}{v} \rceil) \end{aligned}$$

$$R = \frac{A(\sigma)}{OPT(\sigma)} \leq 1 + \frac{k - \lceil \frac{k}{v} \rceil}{\lceil \frac{m \cdot L}{v} \rceil}$$

Together with the results in [13], we have

$$R = \min(2, 1 + \frac{k - \lceil \frac{k}{v} \rceil}{\lceil \frac{m \cdot L}{v} \rceil})$$

We can disregard the $\lceil \quad \rceil$ operation in our analysis when OPT(σ) approach infinity. Then the above formula becomes

$$R = \min(2, 1 + \frac{v-1}{c \cdot L})$$

The specific values of v and L depend on how the capacity of servers are discretized. Since both variables are represented as multiples of items, we can use their ratio t = L / v to represent the average application load in multiples of the server capacity. Then we have

$$R = \min(2, 1 + \frac{1 - \frac{1}{v}}{c \cdot t})$$

Note that 1/v is the load unit. Figure 5 shows how the approximation ratio changes with the parameters. The figure indicates that the approximation ratio is determined mostly by the product of c and t which reflects the total load of all applications in a color set. This ratio is small for reasonably large c and t. It indicates that our algorithm uses a small number of servers to satisfy all application demands when the load is low, and achieves good satisfaction ratio when the load is high. This will be backed up by our experiments in Section 5.

4.3 Practical considerations

In this subsection, we examine some implementation issues that need to be considered in practice.

4.3.1 Server equivalence class

Our algorithm assumes that the physical servers are homogeneous with uniform capacity. (The VMs are obviously heterogeneous just like in Amazon EC2.) This is typical for bin packing algorithms in the literature. [4] One may wonder how realistic this assumption is in practice.

Servers in large data centers are often acquired in large batches where each batch contains a large number of servers with identical hardware. Of course, the servers in a large data center are not all identical. Multiple generations of hardware are likely to co-exist. We can divide the servers into "equivalence classes" based on their hardware settings and run our algorithm within each equivalence class. Our algorithm also assumes that the items have the same unit size. This is not a restriction in practice because the item size is used as the load unit to represent a certain percentage of server capacity. As long as the servers have uniform capacity, the item sizes are identical.

4.3.2 Class constraint

In a cloud computing environment, it is not possible to run every application on every machine due to resource constraint. For e-commerce applications, physical memory is typically the bottleneck resource that limits how many applications a server can run simultaneously. This limit is modeled explicitly in our algorithm as the class constraint and can be enforced by the cloud service provider [17]. Note that it is not a limit imposed by our algorithm. We only need to run enough applications to drive the server busy, e.g., to saturate its CPU cycles. As analyzed previously, our algorithm works well when the aggregate load of applications in a color set is high. For Internet applications, this typically means that the CPU demand relative to the memory consumption is high. For those applications, a fairly conservative class constraint is usually sufficient to saturate the servers. As we will see later in the evaluation, the demand satisfaction ratio of our algorithm is not sensitive to the particular choice of the class constraint used.

Note that we do not require the applications or the VMs encapsulating them to have identical memory requirements. We use the self-ballooning technique in Xen 4.0 to adjust the memory allocation among the VMs [18]. This technique allows the hypervisor to collect the unused memory from the VMs into a shared pool which can then be accessed by the needy guest OS. It also allows the hypervisor to withhold a certain amount of memory as reserve which can be allocated to the VMs in an on-demand manner.

4.3.3 Load change

The load of data center applications can change continuously. We only need to invoke our algorithm periodically or when the load changes cross certain thresholds. The actions in each step of the algorithm are consolidated and executed when the algorithm finishes. Hence, if a flash crowd requires an application to add a large number of servers, all the servers are started in parallel. As we will see later in the simulations, our algorithm is highly efficient and can scale to tens of thousands of servers and applications.

The amount of load change during a decision interval may correspond to the arrivals or departures of several items in a row. A large load unit reduces the overhead of our algorithm because the same amount of load change can be represented by fewer items. It also increases the stability of our algorithm against small oscillation in load. On the other hand, it can lead to inefficient use of server resources and decrease the satisfaction ratio of application demands. The granularity of load change we can capture is limited by the load unit.

4.3.4 Optimization

In our algorithm, each color set has at most one unfilled bin. The number of colors in the unfilled bin sometimes is smaller than the class constraint. When the number of color sets is large, the aggregate unfilled capacity in those bins is non-negligible. The demand satisfaction ratio can be improved if we use the unfilled capacity to satisfy the applications whose demands are not completely satisfied, even if they belong to different color sets. More specifically, we sort the list of unfilled bins with spare color slots in descending order of their capacity. We also sort the list of applications in descending order of their unsatisfied demand. Then we use a greedy algorithm to put those applications into the unfilled bins: when a bin is full or reaches the class constraint, we move on to the next one in the list. By allowing temporary violation of the color set property, we can improve the demand satisfaction ratio when the resources are tight. Note that the class constraint in the original CCBP problem is never violated.

4. A version of the problem called VCCBP allows the bins to be of different sizes. However, the goal there is to minimize the total size of bins used, which is not applicable to our settings.

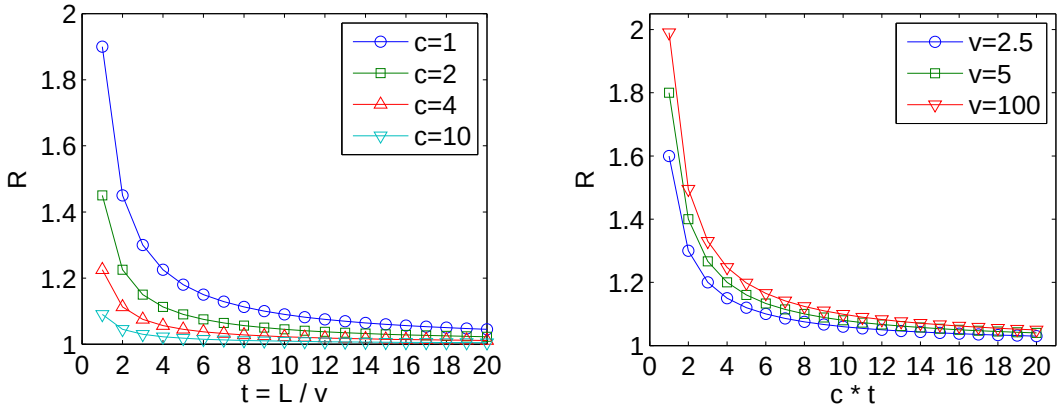


Fig. 5. Approximation ratio as the parameters change

5 Experiments

We evaluate the effectiveness of our system in experiments. The Web applications used in the experiments are Apache servers serving CPU intensive PHP scripts. Each application instance is encapsulated in a separate VM. The servers are connected over a Gigabit ethernet. The client machines run httperf to invoke the PHP scripts on the Apache servers. This allows us to subject the applications to different degrees of CPU load by adjusting the client request rates. We consider a server as "full" when its capacity reaches 80%. This leaves some room for additional load increase as we start some of its applications on other servers. To save time on the experiments, we configure the Application Scheduler with an aggressive two minutes interval between invocations. This allows us to complete the experiments in a timely manner. In practice, we expect the Scheduler to run much less frequently. We configure the algorithm to run no more than four applications on each server (i.e., c=4). The load unit is 1%.

5.1 Load shifting

We first evaluate the effectiveness of load shifting in our algorithm to avoid application placement change. Figure 6 shows the experiment with three servers and three applications. The servers are with Intel E5420 CPU and 8 GB of RAM and run Xen-3.3.1. We keep the scale of the experiment small so that we can present results for all servers. In the next subsection, we will present experiments with a group of 30 servers. Different shades are used to distinguish the applications in figure 6. Application 1 and 3 run on server 1, application 2 and 3 run on server 2, and application 2 runs on server 3. Server 3 is the only server with spare capacity (i.e., the unfilled bin). All applications belong to the same color set. We first increase the load on application 1 gradually. Without load shifting, a new instance of the application has to start on server 3. With load shifting, the algorithm shifts the load of application 3 onto server 2 and gives the released capacity to application 1. Since server 2

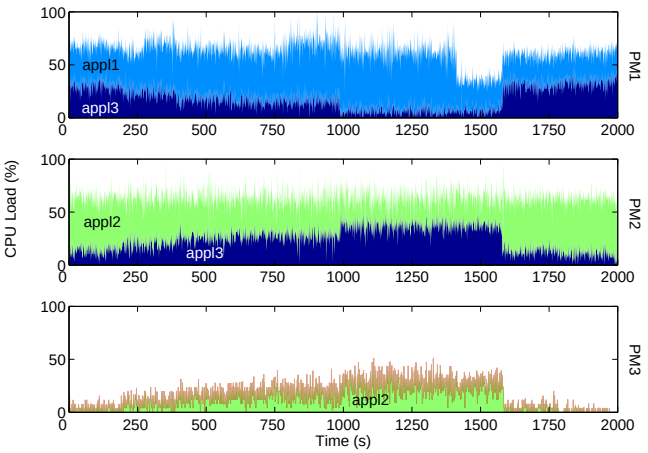


Fig. 6. Load shifting to avoid placement change

is also fully utilized, the algorithm shifts the load of application 2 onto server 3 in order to accommodate the migrated load of application 3 from server 1. As a result, the load on server 3 increases as we increase the load on application 1, even though application 1 does not run on server 3. In other words, we use the spare capacity in the unfilled bin to absorb the demand increase in the full bin.

After it reaches a stable state, we decrease the load on application 1 gradually and the reverse process happens. The algorithm shifts the load on application 3 from server 2 to server 1 in order to keep server 1 full. It shifts the load of application 2 from server 3 to server 2 in order to keep server 2 full. As a result, the load on server 3 decrease in order to fill the gap left by application 1, even though application 1 does not run on server 3.

Note that around 1450 seconds, there is a drop in the load on server 1 for about 100 seconds. Thus the system seems to have two unfilled bins during that period. This is because the decision algorithm executes only periodically. The load decrease on application 1 happens

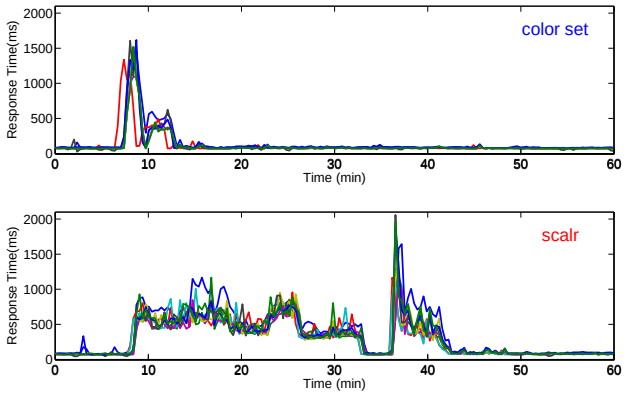


Fig. 8. Comparison of the Response time

shortly after a run of the algorithm has just finished. The Scheduler does not know about the load change until it executes again two minutes later. During this period, the L7 switch continues forwarding the requests according to the existing distribution. Thus a load decrease on application 1 leads to a corresponding load decrease on server 1. As soon as the Scheduler executes again, it uses load shifting to fill the gap.

5.2 Auto scaling

We evaluate the auto scaling capability of our algorithm with nine applications and 30 Dell PowerEdge servers with Intel E5620 CPU and 24 GB of RAM. The servers run Xen-4.0 and Linux 2.6.18. The results are shown in figures 7 and 8. We increase the load of one application dramatically to emulate a "flash crowd" event while keeping the load of the other applications steady. Figure 7 (left) shows the request rate of the flash crowd application and the number of active servers (i.e., APMs) used by all applications over the course of the experiment. Initially, the load in the system is low and only a small number of servers are used. When the flash crowd happens, our algorithm detects the skyrocketing request rate quickly and scales up the server resources decisively. The figure shows that it uses up all 30 servers during the peak demand. Then we reduce the request rate of the application gradually to emulate that the flash crowd is over. The algorithm scales down the server resources accordingly to conserve energy.

In the above experiment, a server on average spends 39% of the time in standby mode due to green computing. This translates into roughly 51 Watts power-saving per server or 1530 Watts for the group of 30 servers used in the experiment.

The middle and the right figures in Figure 7 show the reply rate and the percentage of errors for the flash crowd application during the experiment. Figure 8 shows the response time for all applications. The reply rate indicates how many requests got successful replies. It can be seen as the throughput of the application. As a target for comparison, we use the Scalr open source implementation of auto scaling in Amazon EC2 [17]. Scalr adjusts the number of VM instances for an application on EC2 dynamically based on the observed load. It considers each application individually and has no notion of the "class constraint" . Hence, it can be seen as a "classless" algorithm which does not set a fixed number on how many applications a server can run simultaneously. Of course, its application placement still needs to consider if the candidate server has enough resources to accommodate the application.

As we can see from the figure, at the beginning of the flash crowd, both algorithms experience reduced throughput and increased response time. Both have a certain percentage of requests result in errors. This is because it takes time to start and warm up the new VM instances. However, this period is much shorter for our algorithm than for Scalr: our algorithm restores to the normal QoS in less than five minutes, while Scalr still suffers much degraded performance even after 25 minutes. Auto scaling in Scalr is performed on a per application basis and no load shifting (like that in Figure 6) is performed across the applications. It is possible for some applications to have unsatisfied demands while other application servers still have unfilled capacity.

6 Simulations

The previous section has demonstrated the effectiveness of our system in real experiments. This section evaluates the performance of our application scheduling algorithm in large scale simulation. Our simulator uses the same code base for the scheduling algorithm as the real implementation in the previous section. This ensures the fidelity of our simulation results.

The input to our simulation includes the resource capacity of each server, the memory demand of each application instances, and the total CPU demand of each application in every decision interval. We assume that all servers are homogeneous. We define the "demand ratio" as the ratio between the aggregate demand of all applications and the aggregate capacity of all servers. We denote it as D. For example, D = 0.90 means that the average server utilization will be 90% if all application demand is satisfied. Then according to the class constraint (c) and the memory capacity of the server (C_M), we set the max memory demand of application instances to be $\frac{C_M}{c}$. The total CPU demand is allocated to the set of applications as follows: According to the demand ratio (D) and the total CPU capacity of all servers (C_{CPU}), we can get the the total CPU demand of all applications as $D * C_{CPU}$. Each application selects a random number between 0 and 1 as its weight. Then the CPU demand is allocated proportional to their normalized weights. During each decision interval, the demand of each application changes randomly by 20%.

We define a server as active if it has at least one application instance running. We call it an APM (Active

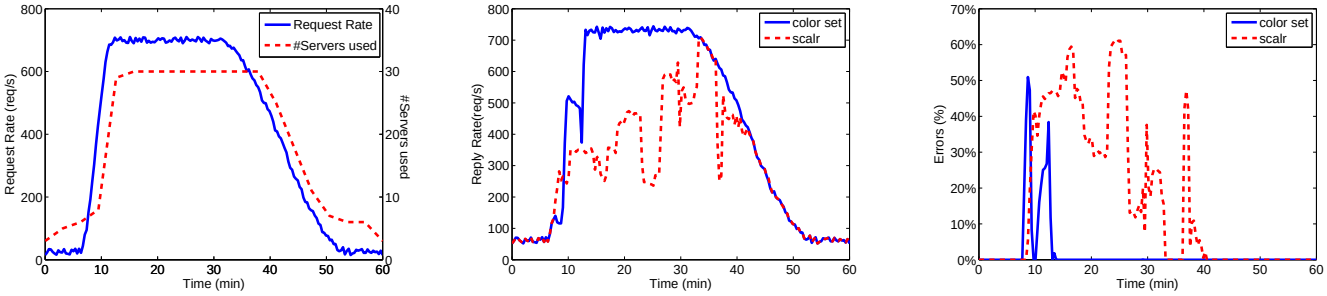


Fig. 7. Left figure: Request rate and #servers used. Middle and right figures: Comparison with Scalr

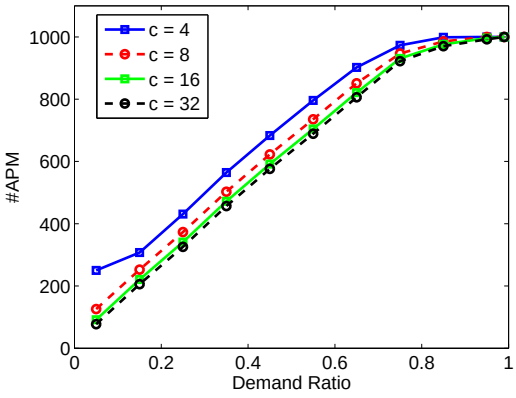


Fig. 9. Number of APMs as the demand increases

Physical Machine). Otherwise, it is inactive and could potentially be turned off to save energy. The simulator invokes our scheduling algorithm according to the prepared input and computes performance metrics such as the number of APMs in the system, the decision time, the application demand satisfaction ratio, and the number of placement changes. Each data point is repeated 200 times and the average is taken.

6.1 Application demand ratio

The first set of experiments are conducted in a group of one thousand servers with one thousand applications. We increase the demand ratio (D) from 0.05 to 0.99. The results are shown in figure 9 and 10. Figure 9 shows that the number of APMs increases with the demand and reaches 100% when the demand is very high. It decreases slightly with the class constraint when the demand is low, but the effect is small. The average decision time in each run of the algorithm is shown in figure 10 (left). Note that the decision time is based on the real execution of the actual code used in the deployment. The figure indicates that the decision time increases with the demand ratio and with the class constraint. This is mostly due to the search time to find a chain of items when the application demand changes as described in Section 4.1.1. The longest decision time is less than 0.8 second, which is very fast for a system of this size. The middle figure shows that the demand satisfaction ratio remains at 100% until D=0.65 and then decreases slightly. Somewhat surprisingly, the class constraint has essentially no impact on the satisfaction ratio: when the class constraint increases from 4 to 32 (i.e., by a factor of 8), the satisfaction ratio remains essentially unchanged. This is due to the optimization in Section 4.3.4 which allows us to satisfy more application demands during high load, even when the class constraint is small. This shows that the performance of our algorithm is not sensitive to the specific choice of the class constraint. The right figure shows the number of placement changes broken down into application starts and stops. Both of them increase with the demand ratio and decreases with the class constraint: when each server can run a larger number of applications simultaneously, there is less need for placement change. This is more profound when the demand is high. There are more application starts than stops because some application instances keep running throughout the simulation. Note that we do not migrate VMs.

6.2 Scalability

We evaluate the scalability of the algorithm by increasing both the number of servers and the number of applications from 1000 to 10,000. Figure 11 (left) shows how the decision time increases with the system size. As we can see from the figure, our algorithm is extremely fast, even when implemented in Python: the decision time is less than 4 seconds when the system size reaches 10,000. The middle figure shows that the demand satisfaction ratio is independent of the system size (depends on D as shown previously). The right figure shows that the number of placement changes increases linearly with the system size. Again the class constraint has a bigger impact when the demand is higher. When averaged over the number of servers, each server experiences roughly a constant number of placement changes for any given demand ratio.

6.3 Application number

Next we vary the ratio between the applications and the servers by increasing the number of applications from

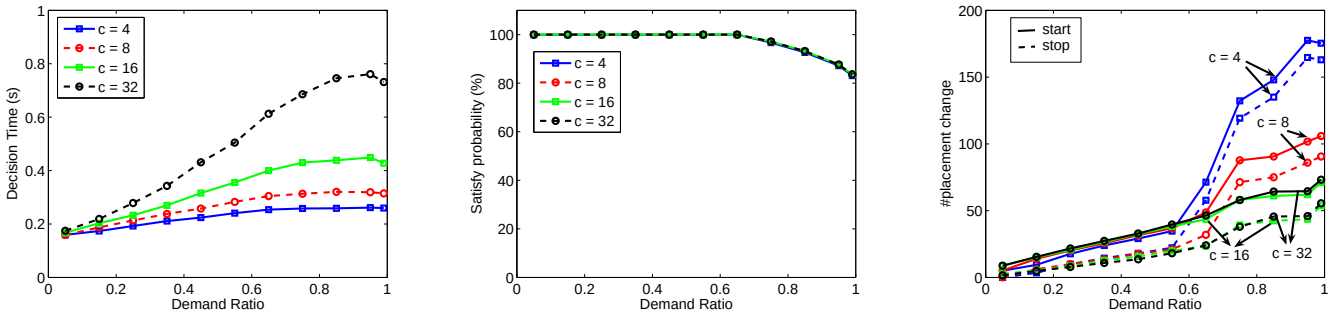


Fig. 10. Performance of the algorithm as the demand increases

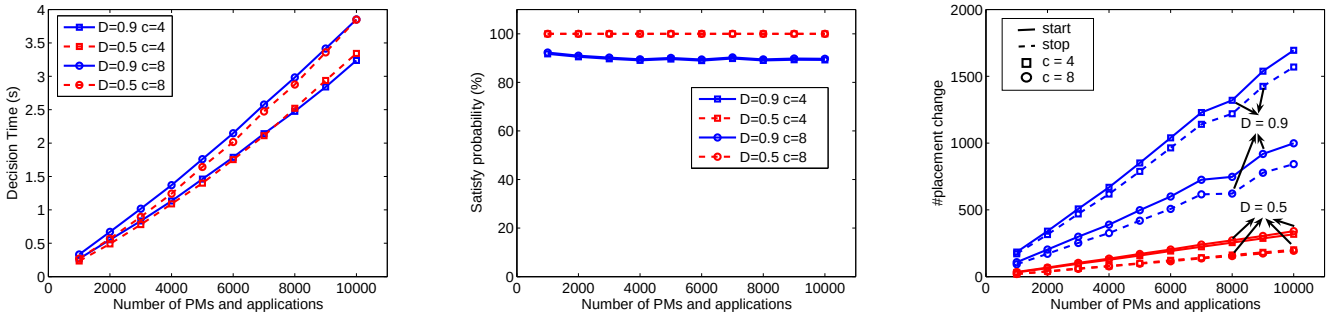


Fig. 11. Scalability of the algorithm as the number of servers and applications increases

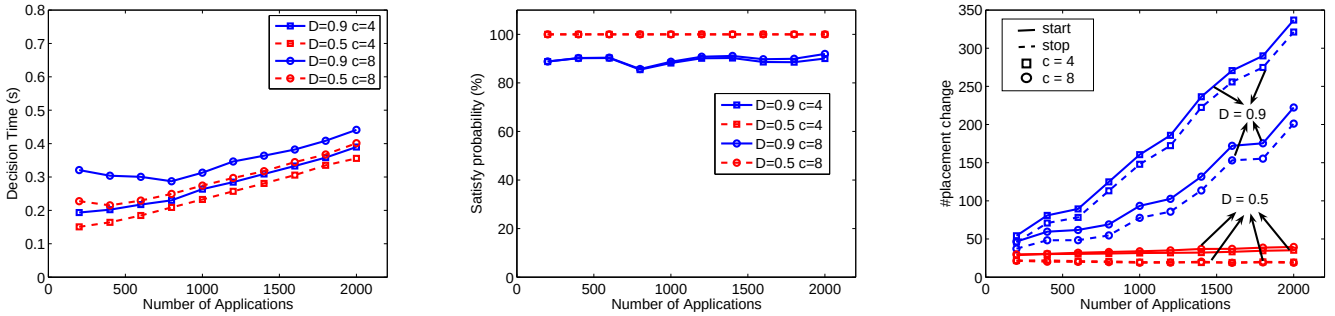


Fig. 12. Performance of the algorithm as the number of applications increases

200 to 2000. The number of servers is fixed at 1000. The results are shown in figure 12. The left figure shows that the decision time is longer (but remains under 0.5 second) when there are more applications and when the class constraint is larger. The middle figure shows that the satisfaction ratio is little affected by the number of applications. The right figure shows that when the demand is high, more applications lead to more placement changes. This is because for a given class constraint, more applications means fewer running instances for each application. Again a larger class constraint helps reduce placement churns. Both of them have a much smaller impact when the demand is lower.

7 Related Work

The traditional bin packing problem has been extensively studied in the literature (see the survey in [19]). The vector bin packing problem considers multi-dimensional constraints when packing items into a minimum number of bins [20]. One may think we can consider the CPU demand and the memory requirement of an Internet application as individual elements in the vector and use vector bin packing to solve our problem. Unfortunately, the memory requirement of Internet applications has to be satisfied as a whole: a major portion of the memory is consumed anyway even when the application receives little load. This is especially true for Java applications whose memory usage may depend on the past load due to garbage collection. Hence, we cannot divide the memory requirement and satisfy it in a piecemeal manner across the servers. None of the existing bin packing problems can be applied in our environment.

The Class Constrained Multiple Knapsack problem

(CCMK) aims to maximize the total number of packed items under the restriction that each knapsack has a limited capacity and a bound on the number of different types of items it can hold [21], [22]. Unlike CCBP, it does not attempt to minimize the number of knapsacks used. Hence, unlike our algorithm, it does not support green computing when the system load is low.

A number of approximation algorithms have been developed for CCBP. Most of them are offline algorithms which do not support item departure. The rest are strict online algorithms which do not allow movements of already packed items. In the case of item departure, the departed item is removed but the rest of the items in the bins are not re-packed. When a color set becomes unfilled due to application leaves, those algorithms do not maintain the property that there is at most one unfilled color set in the system. This can degrade the performance severely because each color set is packed independently. It has been shown that the existing color set algorithms perform poorly in the face of frequent item departure [13]. They cannot be applied in a cloud computing environment where the application demands change dynamically.

Resource provisioning for Web server farms has been investigated in [23], [24], [25], [26]. Some allocate resources in the granularity of whole servers which can lead to inefficient resource usage. Some do not consider the practical limit on the number of applications a server can run simultaneously [25]. Bhuvan *et al.* support shared hosting, but manage each application instance independently [23]. They do not provide the auto-scaling property. Mohit *et al.* group applications into service classes which are then mapped onto server clusters [24]. However, they do not attempt to minimize the placement changes when application demands vary and is mostly for offline use. Zhang *et al.* organize a set of shared clusters into a network and study resource allocation across shared clusters [26], which is not the focus of this paper.

Process migration has been studied in various contexts, e.g., [27]. Unlike virtualization technology, it does not capture the execution environment of the running processes. Nor does it support the auto scaling of the processes based on the observed demand.

Application placement in enterprise environments has been studied in [28], [29], [30], [31]. They run multiple applications on the same set of servers directly without using VMs or Sandbox. Their approach is suitable when the applications are trustworthy (e.g., enterprise applications). It is not suitable for a cloud environment where applications come from untrusted users. Unlike ours, their decision algorithm has no concern on green computing and is based on a set of heuristics with no provable bounds or optimality. Our algorithm can scale to an order of magnitude more servers than those in [28], [29] because the complexity of our algorithm is much lower.

Like our system, the Google AppEngine service provides automatic scaling for Web applications. The users are charged by the CPU cycles consumed, not by the number of application instances. Its internal algorithm used is not disclosed. Our algorithm potentially can be used to implement such a service. The applications in AppEngine must run inside a sandbox with severe restrictions on what they can do. At the time of this writing, it supports mostly applications written in Java and Python [5] or Google's own Go programming language. This makes it difficult to port legacy applications onto their platform. In contrast, porting an existing application onto our VM platform is much easier. It gives the users great flexibility in choosing their favorite programming languages, operating systems, libraries, etc..

There are also some cloud vendors providing auto-scaling solutions for cloud users (see the survey in [32]). Users are allowed to define a set of rules to control the scaling actions. However, the rules and the load balancing strategies they used are very simple. Just like the Scalr in Amazon EC2 [17], they perform the scaling actions simply when some conditions are met and balance the load evenly across all instances. Since they do not take the state of the whole system into consideration, they cannot reach a globally optimal decision.

8 Conclusions and Future Work

We presented the design and implementation of a system that can scale up and down the number of application instances automatically based on demand. We developed a color set algorithm to decide the application placement and the load distribution. Our system achieves high satisfaction ratio of application demand even when the load is very high. It saves energy by reducing the number of running instances when the load is low.

There are several directions for future work. Some cloud service providers may provide multiple levels of services to their customers. When the resources become tight, they may want to give their premium customers a higher demand satisfaction ratio than other customers. In the future, we plan to extend our system to support differentiated services but also consider fairness when allocating the resources across the applications. We mentioned in the paper that we can divide multiple generations of hardware in a data center into "equivalence classes" and run our algorithm within each class. Our future work is to develop an efficient algorithm to distribute incoming requests among the set of equivalence classes and to balance the load across those server clusters adaptively. As analyzed in the paper, CCBP works well when the aggregate load of applications in a color set is high. Another direction for future work is to extend the algorithm to pack applications with complementary bottleneck resources together, e.g., to co-locate a CPU intensive application with a memory intensive one so

5. The applications must be written purely in Python without any C extension.

that different dimensions of server resources can be adequately utilized.

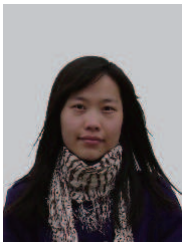
ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work is supported by National Natural Science Foundation of China Project 61170056.

REFERENCES

[1] "Amazon elastic compute cloud (Amazon EC2), http://aws.amazon.com/ec2/."
[2] A. Cohen, S. Rangarajan, and H. Slye, "On the performance of tcp splicing for url-aware redirection," in *Proc. of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, 1999.
[3] "WebLogic, http://www.oracle.com/appserver/weblogic/weblogic-suite.html."
[4] "Google App Engine, http://code.google.com/appengine/."
[5] M. Armbrust *et al.*, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.
[6] L. Siegele, "Let it rise: A special report on corporate IT," in *The Economist*, Oct. 2008.
[7] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proc. of the ACM Symposium on Operating System Principles (SOSP'01)*, Oct. 2001.
[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP'03)*, Oct. 2003.
[9] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An extensible framework for managing clusters of virtual machines," in *Proc. of the Large Installation System Administration Conference (LISA'07)*, Nov. 2007.
[10] J. Zhu, Z. Jiang, Z. Xiao, and X. Li, "Optimizing the performance of virtual machine synchronization for fault tolerance," *IEEE Transactions on Computers*, Dec. 2011.
[11] "RUBiS, http://rubis.ow2.org/."
[12] "Linux Documentation, http://www.kernel.org/doc/documentation/power/states.txt."
[13] H. Shachnai and T. Tamir, "Tight bounds for online class-constrained packing," *Theor. Comput. Sci.*, vol. 321, no. 1, pp. 103–123, 2004.
[14] L. Epstein, C. Imreh, and A. Levin, "Class constrained bin packing revisited," *Theor. Comput. Sci.*, vol. 411, no. 34-36, pp. 3073–3089, 2010.
[15] E. C. Xavier and F. K. Miyazawa, "The class constrained bin packing problem with applications to video-on-demand," *Theor. Comput. Sci.*, vol. 393, no. 1-3, pp. 240–259, 2008.
[16] M. R. Garey and D. S. Johnson, "A 71/60 theorem for bin packing," *Journal of Complexity*, vol. 1, 1985.
[17] "Scalr: the auto scaling open source Amazon EC2 effort, https://www.scalr.net/."
[18] D. Magenheimer, "Transcendent memory: A new approach to managing RAM in a virtualized environment," in *Linux Symposium*, 2009.
[19] G. Galambos and G. J. Woeginger, "On-line bin packing-a restricted survey," *Physica Verlag*, vol. 42, no. 1, 1995.
[20] C. Chekuri and S. Khanna, "On multidimensional packing problems," *SIAM J. Comput. Issue 4*, vol. 33, 2004.
[21] H. Shachnai and T. Tamir, "Noah's bagels-some combinatorial aspects," in *Proc. 1st Int. Conf. on Fun with Algorithms*, 1998.
[22] ——, "On two class-constrained versions of the multiple knapsack problem," *Algorithmica*, vol. 29, no. 3, pp. 442–467, 2001.
[23] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 239–254, 2002.
[24] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: a mechanism for resource management in cluster-based network servers," *SIGMETRICS Perform. Eval. Rev.*, vol. 28, no. 1, pp. 90–101, 2000.
[25] J. L. Wolf and P. S. Yu, "On balancing the load in a clustered web farm," *ACM Trans. Internet Technol.*, vol. 1, no. 2, pp. 231–261, 2001.
[26] C. Zhang, V. Lesser, and P. Shenoy, "A Multi-Agent Learning Approach to Online Distributed Resource Allocation," in *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009.
[27] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, 2002.
[28] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications," in *Proc. of the International World Wide Web Conference (WWW'06)*, May 2006.
[29] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proc. of the International World Wide Web Conference (WWW'07)*, May 2007.
[30] C. Adam and R. Stadler, "Service middleware for self-managing large-scale systems," *IEEE Transactions on Network and Service Management*, vol. 4, no. 3, pp. 50–64, 2007.
[31] J. Famaey, W. D. Cock, T. Wauters, F. D. Turck, B. Dhoedt, and P. Demeester, "A latency-aware algorithm for dynamic service placement in large-scale overlays," in *Proc. of the IFIP/IEEE international conference on Symposium on Integrated Network Management (IM'09)*, 2009.
[32] E. Caron, L. Rodero-Merino, F. Desprez, and A. Muresan, "Auto-scaling, load balancing and monitoring in commercial and open-source clouds," INRIA, Rapport de recherche RR-7857, Feb. 2012.

Zhen Xiao is a Professor in the Department of Computer Science at Peking University. He received his Ph.D. from Cornell University in January 2001. After that he worked as a senior technical staff member at AT&T Labs - New Jersey and then a Research Staff Member at IBM T. J. Watson Research Center. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of ACM and IEEE.



Qi Chen received bachelor's degree from Peking University in 2010. She is currently a doctoral student at Peking University. Her current research focuses on the cloud computing and parallel computing.

Haipeng Luo received bachelor's degree from Peking University in 2011. He is currently a doctoral student at Princeton University. His current research focuses on machine learning.