Practical Resource Provisioning and Caching with Dynamic Resilience for Cloud-Based Content Distribution Networks

Menglan Hu, Jun Luo, *Member, IEEE*, Yang Wang, and Bharadwaj Veeravalli, *Senior Member, IEEE*

Abstract—Content distribution networks (CDNs) built on clouds have recently started to emerge. Compared to conventional CDNs, cloud-based CDNs have the benefit of cost efficient hosting services without owning infrastructure. However, resource provisioning and replica placement in cloud CDNs involve a number of challenging issues, mainly due to the dynamic nature of demand patterns. To deal with this dynamic nature, this paper proposes a set of novel algorithms to solve the joint problem of resource provisioning and caching (i.e., replica placement) for cloud-based CDNs with an emphasis on handling the dynamic demand patterns. Firstly, we propose a provisioning and caching algorithm framework called *Differential Provisioning and Caching* (DPC) algorithm, which aims to rent cloud resources to build CDNs and whereby to cache contents so that the total rental cost can be minimized while all demands are served. DPC consists of 2 steps. Step 1 first maximizes total demands supported by unexpired resources. Then, step 2 minimizes the total rental cost for new resources to serve all remaining demands. For each step we design both greedy and iterative heuristics, each with different advantages over the existing approaches. Moreover, to dynamically adjusts the placement of contents and route maps, we further propose the *Caching and Request Balancing* (CRB) algorithm, which is light-weight and thus can be frequently executed as a companion of DPC to maximize the total demands. Performance evaluation results are presented to demonstrate the effectiveness and competitiveness of our approaches when compared to existing algorithms.

Index Terms—Cloud computing, content distribution, resource provisioning, replica placement, request routing.

1 Introduction

With the successful deployment of commercial systems and increasing user popularity, content distribution networks (CDNs) have received much attention in recent years. Conventional CDNs such as Akamai built hundreds of data centers to distribute the contents across the world. It has hence become financially prohibitive for small content providers to compete on a large scale by deploying new data centers.

The emerging cloud vendors such as Amazon S3 are creating new opportunities to enable cost-effective CDNs. As the cloud vendors provide on-demand and cost-effective content storage and delivery capabilities, one can build CDNs upon the clouds without investments on installing and maintaining the infrastructure while providing scalable service. As a customer, a cloud CDN may benefit from elastic cloud charge models. Also, the cloud CDN can dynamically adjust the leases of bandwidth, virtual machine (VM), and storage resources based on the runtime demand rates to reduce the total rental costs without severely sacrificing the service performance.

There have been some initial attempts in leveraging clouds to support content distribution. For example, Netflix moved its streaming servers, data stores, and other customer-oriented APIs to Amazon Web Services (AWS) [2]. In academia, several recent works [3], [4], [5], [10] also studied the problems of building CDNs on clouds. However, these papers either relied on over-simplified assumptions that storage and/or bandwidth capacities are infinite or only one content is considered [3], [4], [10], while others required to solve linear programming problems [5], which may be impractical for large-scale systems with rapidly varying demand patterns. Therefore, it is desirable to design efficient algorithms which are practical to large scale cloud CDNs with finite resources.

In addition, it is well known that the problems of resource provisioning and replica placements in cloud CDNs are notoriously difficult, mainly due to the dynamics and diversities in users' request patterns, which lead to the following so-called *update dilemma* that is commonplace but more or less neglected in the above mentioned studies.

The dilemma includes several aspects. Firstly, due to rapidly varying demand patterns, these algorithms need frequently update provisioning and caching solutions to keep up with the dynamic variation (e.g., [10] suggested a frequency of every 10 minutes in experiments). Unfortunately, this eager solution is not always cost-effective when considering the dynamic nature of the demands. Usually, after being leased, a cloud server has to spend a significant amount of time in initializing its configuration before it is ready to use (e.g. ephemeral disk problem [21]). During this period, if the demand rates or the access patterns are rapidly changed, it is always hard, if not impossible, for the algorithms to react in time by frequently updating the solutions and leasing new resources because of

- *M. Hu and J. Luo are with the School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798. E-mail: {mlhu, junluo}@ntu.edu.sg.*
- *Y. Wang is with the Faculty of Computer Science, University of New Brunswick, Fredetricton, Canada, E3B 5A3. E-mail: ywang8@unb.ca.*
- *B. Veeravalli is with the Department of Electrical and Computer Engineering, National University of Singapore, 4 Engineering Drive 3, Singapore 117576. E-mail: elebv@nus.edu.sg.*

the considerable initialization overhead. It is likely that when a server becomes ready after the initialization phase, demand rates might fall down again, leaving the newly rented server to be under-utilized, which is apparently not cost effective. Additionally, the update operation is also not free and may incur considerable overheads as existing placements are potentially influenced. Another factor restricting the frequent update is that most cloud providers impose a minimum time unit for leasing resources (e.g. 1 hour for Amazon EC2). This further deteriorates the dilemma. For example, once the demand rate arriving at a certain cloud site unexpectedly decreases, the excessively rented resources at that site may be wasted until the current lease period is over. Therefore, in practice it may be unwise to frequently update the solutions to match the runtime demand patterns while less frequent updates could also cause resource wastage and performance degradation.

Clearly, this dilemma stems from the mismatch of the highly unpredictable demand patterns and relatively inflexible resource provisioning, and imposes great challenges to cloud CDNs. Therefore, to maximize the served demands with the provisioned resources, we should consider jointly both the resource provisioning and content caching together with their interactions at the same time for the design of adaptive and practical management algorithms as it has been shown that any unilateral solution could result in the degraded performance and inefficient resource utilization [3], [4], [10].

With these challenges in mind, in this paper we contribute a set of novel algorithms to the joint problem of resource provisioning and caching (i.e., replica placement) in cloud-based CDNs with an emphasis on resolving the mismatch in the update dilemma. Our approaches consist of both long term (i.e., less frequently used) and short term (i.e., frequently used) algorithms, each with its own goals and strategies. This design not only simplifies the algorithms in each phase but also isolates the performance problems for easy analysis.

We first propose a long term provisioning and caching algorithm framework called *Differential Provisioning and Caching* (DPC) algorithm, which rents cloud resources to build the CDNs and thereafter cache contents in such a way that the total rental cost can be minimized while all demands are served. With different focuses in consideration, DPC performs two steps to deal with the time granularity of the lease unit. Step 1 is to maximize the total demands supported by the unexpired resources while Step 2, as a complementary action, is to minimize the total rental cost for the new resources to serve all remaining demands. These steps are not redundant; rather, they cooperate with each other to reach the final goal.

As the formulated problem is NP-complete, for each step of DPC, we design both efficient greedy and iterative heuristics, each having different advantages in achieving the goals. In particular, we develop two greedy heuristics, *Site First Greedy* (SFG) and *Set Cover Greedy* (SCG), each emphasizing a different facet of the problem. SFG is our original contribution while SCG is a variant of the classical greedy heuristic of the *set cover problem* [17]. As the name indicates, SFG prioritizes the content sites first for the content placements to optimize resource utilization and thus we use SFG in step 1 of DCP. We then adopt SCG in step 2 because it is beneficial in making cost-efficient content placements.

Although these simple greedy algorithms are efficient, the quality of the solutions may not be always satisfactory in some case that the resource utilization is a pragmatic concern. Hence, we also design a novel iterative algorithm, referred to as *Utilization-Aware Greedy Search Algorithm* (UAGSA). UAGSA searches for the efficient provisioning/caching solutions by iteratively utilizing SCG/SFG to construct the feasible solutions that can improve resource utilization in CDN sites. Based on SFG, SCG, and UAGSA, we develop two combined algorithms, SFG-UAGSA, and SCG-UAGSA, which are used in steps 1 and 2 of DPC, respectively.

To address the dynamic demand patterns, we also propose the short term *Caching and Request Balancing* (CRB) algorithm. By leveraging the previous runs of DPC, CRB dynamically adjusts the content placements and route tables such that the total demands supported by the provisioned resources can be maximized. Since no provisioning is involved and only dynamic adjustments are made, CRB is light-weight and thus can be frequently executed as in conjunction with DPC as a reaction to the dynamically changed demand patterns. Clearly, by combining the two algorithms, DCP and CRB, the update dilemma could be naturally solved.

To the best of our knowledge, our paper is the first attempt to realistically deal with the dynamic nature in users' demand patterns on cloud-based CDNs. The proposed algorithms are shown to be efficient in terms of both performance (i.e., amount of served requests) and rental cost under rapidly varying demand patterns via simulation studies.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces mathematical models, assumptions, and problem formulation. Section 4 proposes the provisioning and caching algorithms including DPC, SFG, SCG, and UAGSA. Section 5 describes the CRB algorithm. Section 6 presents simulation results to evaluate the algorithms, with conclusions following in Section 7.

2 Related Work

A number of replica placement algorithms for content distribution have been proposed in the literature. In terms of minimizing content retrieval cost only, Li et al. [6] and Krishnan et al. [7] showed that replica placement in general network topologies is NP-complete and provided optimal solutions for tree topologies. Kalpakis et al. [8] considered read, write and storage costs in content distribution and presented solutions for tree topologies. Further, Borst et al. [9] developed distributed cooperative cache management algorithms that aimed to minimize bandwidth costs. Under symmetric assumptions on systems and demands, the optimal solution of the linear program is shown to have a rather simple structure. Based on this observation, the authors designed low-complexity cache management and replacement algorithms. Dai et al. [11] aimed to maximize total supported demands in a cooperative manner in all levels of hierarchical cache topologies for IPTV systems. Based on the understanding of real IPTV systems, the authors proposed collaborative caching strategies and corresponding dynamic request routing mechanisms for hierarchical topologies. In another work [13], Applegate et al. applied Lagrangian

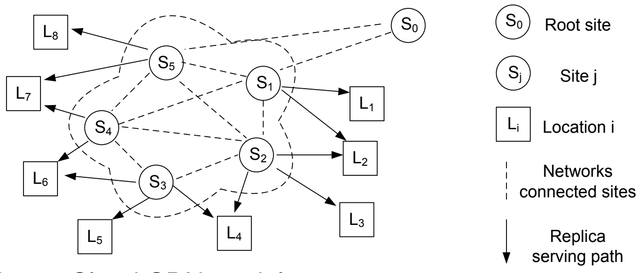


Fig. 1. Cloud CDN model.

relaxation and subgradient methods to solve content placement optimization for large-scale VoD systems.

Some previous studies have considered quality-of-service (QoS) by requiring all user requests to reach replica servers within a certain network distance. Tang et al. [15] proposed QoS-aware algorithms to optimize total storage and update cost. They assumed that requests can be issued from any node, and ignored retrieval cost. Rodolakis et al. [16] added server capacity limitation to the formulation while optimizing storage and retrieval cost. Similar to [15], [16], our paper also considers QoS constraints for replica placement in CDNs, but we step further to consider CDNs built on clouds.

A recent trend of building CDNs on clouds have also started to emerge. Niu et al. [10] studied bandwidth provisioning and multiplexing for video-on-demand (VoD) applications; however, storage and replica placement are neglected in this work. Chen et al. [3] proposed to build CDNs based on storage clouds. Given the NP-hardness of the problem, the authors presented offline and online heuristics. Another work [4] proposed a cloud-assisted live media streaming framework to facilitate a migration of media streaming services to clouds. The authors designed depth first search mechanisms to accommodate temporal and spatial diversities. In [15], [16], [3], [4], storage and/or bandwidth capacities were assumed to be infinite and only one content was to be placed in CDNs. Such over simplified assumptions may restrict the applicability of these studies in real CDNs or cloud CDNs. In contrast, our paper considers placing many contents in cloud-based CDNs with limited storage and bandwidth resources.

Another work close to ours is [5], which solved one-shot content placement optimization problem by relaxing formulated integer programs to linear programs and using Lagrangian relaxation and subgradient methods. They also proposed an online algorithm to adjust the one-shot optimization results. Due to the high computational complexity in Lagrangian relaxation, subgradient methods and linear programming, the solutions in [5] are improper for the problem discussed in this paper. In contrast to [5], our paper designs efficient heuristics, which are more practical for large CDN systems and more dynamic resilient to rapidly varying demand patterns. In addition, as illustrated in Section 1, the algorithms presented by the above papers [3], [5], [4] may not work well due to the practical challenges brought by the dynamic demand patterns. As opposed to these studies, this paper presents novel algorithms to realistically deal with the dynamics.

Parameter	**Definition**
B_j	bandwidth/VM capacity of site j
B_j^r	rest amount of bandwidth/VM capacities at site j
b_j^a	amount of bandwidth/VM capacities unexpired
d_{ik}	input demands of content k at location i
d_{ik}^r	unallocated amount of d_{ik}
d_{jk}^s	amount of demands servable by placing content k on site j
M	a set of locations where users reside
N	a set of sites
N'	a temporary set containing the sites
K	a set of contents provided by the root site
p_j^s	unit storage rental price of site j
p_j^b	unit bandwidth/VM rental price of site j
Q_{ij}	binary parameter indicating whether request can be routed from location i to site j
S_j	storage capacity of site j
S_j^r	rest amount of storage at site j
s_j^a	amount of storage unexpired
Variable	**Definition**
b_j^n	amount of bandwidth/VM capacities newly rented
r_{ijk}	ratio of demands on content k routed from location i to site j
r_{i0k}	ratio of demands on content k routed from location i to the root
s_j^n	amount of storage that should be newly rented
x_{jk}	binary indicating whether content k is cached at site j

TABLE 1
Notation and terminology

3 Problem Formulation

In our settings, a root site aims to serve K (types of) contents (e.g., videos) to users residing on M locations, as shown in Fig. 1. Since the root site is far from the users, to meet QoS constraints, the root needs a CDN to help serve users' demands (or requests). The CDN is built on resources leased from N cloud sites located across the Internet.

Operating a cloud-based CDN on these sites requires 3 kinds of resources: bandwidth, VM, and storage. Accordingly leasing the resources incurs 3 kinds of costs: bandwidth cost, VM rental cost, and storage cost. For simplicity, we assume that all contents are of the same size. This is also a reasonable assumption since servers divide videos into small portions with the same size for the convenience of caching. The request for a long video can also be divided into many requests for small portions. This means that the portions for a video can independently exist in different servers and can be independently served to users [12]. Notice that when considering use patterns (e.g., chunks in one video are served one-by-one), one may further improve the caching algorithm, but this has been beyond the scope of this paper. Accordingly the caching algorithm independently address each file and such assumptions can effectively simplify the problem. Therefore, a fixed-size portion is the minimum unit considered in the paper. In this case, we can assume that serving each request of any content consumes the same amount of resources for both bandwidth and VM. Consequently, we can simply use one parameter to denote both bandwidth and VM resource: let B_j be the bandwidth/VM capacity (in amount of requests that can be served per second) of site j ($j \in N$). Similarly, let p_j^b be the unit price of site j to rent both bandwidth and VM capacities, i.e., cost for renting bandwidth and VM resources that can serve one request per second. Also, let S_j be the storage capacity (in the number of contents that can be stored) of site j and p^s_j be the unit storage price of site j.

Each request, issued by a user, attempts to access a certain content $k \in K$. The content can be video segments in IPTV systems or file blocks in file downloading systems. In order to satisfy QoS requirements for end user requests, we use a binary parameter Q_{ij} to denote whether request can be directed from location i to site j. We let $Q_{ij} = 1$ if requests can be routed from location i to site j. Then we can determine Q_{ij}s beforehand via response time measurement. If proper sites that can meet QoS requirements exist, users' requests can be directed to the proper sites. Otherwise, some requests may be routed to the root as the CDN fails to serve them while meeting QoS requirements. Upon receiving a request, a site returns the corresponding content if bandwidth/VM capacities allow. When the site becomes overloaded, excessive requests will also be routed to the root. Notice that an alternative policy to handle the excessive requests is to redirect them to neighbor sites. Since this alternative policy influences the proposed algorithms little, this policy is not adopted in our models for simplicity. If a request is processed by the root, then the QoS guarantee will probably be violated as the quality distance between the root and any location is too large. Clearly, one of our goals is to maximize the number of requests that are served by the cloud CDN.

Suppose that time is slotted into equal intervals. Provisioning and caching solutions are periodically updated for each interval. As an individual time interval $[t, t + \Delta t)$ is considered, without loss of generality, we drop subscript t in our notations. We define the amount of input demands on content k originated at location i as d_{ik} given by a demand predictor [14] that can predict demands in the coming period $[t, t+\Delta t)$ before time t. We also assume that demand patterns does not vary in each update period Δt. This is a reasonable assumption when Δt is small.

Most cloud providers have a minimum unit time T for the duration of leasing a server (e.g. 1 hour for Amazon EC2). It is possible that the period of updating provisioning solutions (Δt) is shorter than T [10]. In this case, when updating the solutions, only parts of leased resources have expired. Hence, before renting new resources, we should first utilize the currently unexpired resources. Otherwise such resources will be wasted. For the convenience of periodically updating provisioning solutions, we assume that the minimum rental duration for any bandwidth/VM/storage resource is $T = n\Delta t$ where n is a positive integer. That is, provisioning and caching solutions are updated n times during one rental period T. After being leased, each resource will be available for n consecutive time intervals. At the end of each interval, parts of previously rented resources automatically expire and we can rent additional new resources for next n intervals. Fig. 2 shows the timing diagram of an example scenario, wherein $T = 3\Delta t$. The current time is $t + 3\Delta t$. In each execution of the DPC algorithm, resources rented in the latest 2 runs are still unexpired.

Let b^a_j and s^a_j be the amounts of bandwidth/VM capacity and storage capacity that have been rented but have not expired. Let b^n_j and s^n_j be the amounts of bandwidth/VM capacity and storage capacity that should be newly rented for next n time intervals. The storage decision variable $x_{jk} \in 0, 1$ is used to denote whether or not content k is placed in site j. Accordingly x_{jk} represents replica placement strategies. The notation r_{ijk} is used to denote the probability (or fraction) of type-k requests from location i being directed to site j. Hence r_{ijk} actually indicate the request routing strategy. Once available resources are not enough to serve all demands, excessive demands are served by the root and incur performance degradation. Since cloud vendors provide limited resources to each cloud resource user (i.e., the root), this case will happen if a huge amount of demands are requested. We denote the root as site 0 and use r_{i0k} denote the portion of demands that are routed to the root. Demands routed to the root should suffer a large cost p^b_0 as punishment.

The joint resource provisioning and replica placement problem is formulated as an integer programming problem aiming to minimize the total rental cost while all demands are satisfied over time interval Δt as follows:

$$\min \sum_{j\in N} p^b_j b^n_j + \sum_{j\in N} p^s_j s^n_j + \sum_{k\in K}\sum_{i\in M} p^b_0 d_{ik} r_{i0k} \tag{1}$$

s.t.

$$b^n_j + b^a_j \le B_j \quad \forall j \in N \tag{2}$$

$$\sum_{i\in M}\sum_{k\in K} r_{ijk} d_{ik} \le b^n_j + b^a_j \quad \forall j \in N \tag{3}$$

$$s^n_j + s^a_j \le S_j \quad \forall j \in N \tag{4}$$

$$\sum_{k\in K} x_{jk} \le s^n_j + s^a_j \quad \forall j \in N \tag{5}$$

$$r_{ijk}(1 - Q_{ij}) = 0 \quad \forall i \in M, j \in N, k \in K \tag{6}$$

$$0 \le r_{ijk} \le x_{jk} \quad \forall i \in M, j \in N, k \in K \tag{7}$$

$$x_{jk} \in \{0, 1\} \quad \forall j \in N, k \in K \tag{8}$$

$$r_{i0k} + \sum_{j\in N} r_{ijk} = 1 \quad \forall i \in M, k \in K \tag{9}$$

Constraints (2) - (5) capture the bandwidth/VM limit and disk limit at each site, respectively. Constraint (6) then guarantees that the QoS requirement can be satisfied. It is possible that not all links satisfy QoS requirements. Constraint (7) captures the fact that site j can serve content k only when it has a copy locally. Constraint (8) states that we always store either the entire content or none of it at a site. Constraint (9) guarantees that the total fraction of requests served is 1.

The formulation includes a set cover problem when b^a_j and s^a_j are equal to zero, which is a famous NP-complete problem [20]. Since the solutions are periodically updated (e.g., in every 30 minutes), we need effective heuristics which are practical for large CDN systems. In this case, the following sections present a set of efficient heuristics. Finally, we list the notation and terminology in Table 1.

4 Provisioning and Caching Algorithms

To deal with the minimum server lease time, we first propose a two-step algorithm framework, the *Differential*

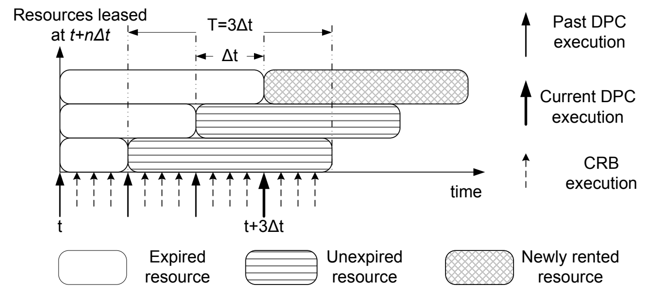


Fig. 2. Timing diagram of DPC and CRB.

Algorithm 1 Greedy Heuristic (SFG/SCG)

```
1:  $N' \leftarrow N$
2:  while true do
3:      Select a pair $(j,k)$ by SFG or SCG pair selection
4:      if no pair is found then
5:          Return
6:      else
7:          for each location $i$ in site $j$'s coverage do
8:              if $d_{ik} > 0$ then
9:                  $d \leftarrow \min(d^r_{ik}, B^r_j)$
10:                 $r_{ijk} \leftarrow r_{ijk} + d/d_{ik}$
11:                 $r_{i0k} \leftarrow r_{i0k} - d/d_{ik}$
12:                 $d^r_{ik} \leftarrow d^r_{ik} - d$
13:                 $B^r_j \leftarrow B^r_j - d$
14:                 $S^r_j \leftarrow S^r_j - 1$
15:             end if
16:         end for
17:     end if
18: end while
```

Provisioning and Caching (DPC) algorithm. The intuition is that to minimize the total cost, we first maximize the demands supported by the unexpired resources in step 1. Then, step 2 minimizes the total cost of renting new resources such that all remaining demands can be served. For each step, we propose greedy and iterative heuristic algorithms, as detailed in the following subsections. These heuristics assign content placements. Provisioning solutions can be finally determined by content placement decisions.

4.1 Greedy Heuristics

In this subsection we describe 2 greedy heuristics, *Site First Greedy* (SFG), and *Set Cover Greedy* (SCG), which share the same algorithm framework shown in Algorithm 1. We use a data structure "content-site pair" specified by a pair (j,k) to denote the placement of content k on site j. Let d^s_{jk} be the amount of demands that can be served by placing content k on site j. A pair (j,k) can be selected only if $d^s_{jk} > 0$. The greedy heuristics work by repeatedly selecting pairs and implementing the pairs (adds the selected pairs to the solution set). Each greedy heuristic has its own policy for selecting pairs (Line 3 in Algorithm 1), as detailed in Sections 4.1.1 and 4.1.2. Algorithm 1 terminates if no pair can be found. Otherwise, the selected pair (j,k) can be implemented by placing content k at site j and updating the routing map, i.e., r_{ijk} and r_{i0k}.

Algorithm 2 SFG Pair Selection

```
1:  while $N' \neq \emptyset$ do
2:      Select a site $j$ with the largest $\gamma_j$ from $N'$
3:      Select a content $k$ from $K$ such that $d^s_{jk}$ is maximized
4:      if $d^s_{jk} = 0$ then
5:          $N' \leftarrow N' - j$
6:      else
7:          Return pair $(j,k)$
8:      end if
9:  end while
10: Return Null
```

In Algorithm 1, N' is a temporary set containing all sites and will be used by SFG (Algorithm 2); d is a temporary variable denoting the allocated (or migrated) demands in one placement operation; d^r_{ik} is the unallocated amount of d_{ik}; B^r_j is the rest amount of capacity at site j and S^r_j is the rest amount of storage at site j. The time complexity of Algorithm 1 is $O(KMN)$ because at most KN pairs may be selected and for placing a pair at most M locations are searched. In practice, M and N are rather small when compared to K and thus K contribute the most to the time complexity.

4.1.1 Site First Greedy (SFG) Pair Selection

As shown in Algorithm 2, this heuristic selects a content-site pair in two steps. It first selects a site with the largest γ_j value given as follows:

$$\gamma_j = \frac{B^r_j}{S^r_j} \tag{10}$$

This metric gives high priorities to the sites with comparatively abundant resources, i.e., the sites with high bandwidth/VM capacities and low storage. Then it assigns the site with the content whose total demand can be served by the site in its coverage is the largest. The intuition of this policy is to optimize resource utilization by first selecting the sites with comparatively abundant resources. This is because popular contents can efficiently consume the resources of such sites. If such sites are not given high priorities, the bandwidth/VM resources of these sites may be wasted because unpopular contents may not efficiently utilize the bandwidth/VM resources of these sites. Since SFG can efficiently utilize available resources, it has the advantage of maximizing supported demands and can be adopted in the first step of DPC.

4.1.2 Set Cover Greedy (SCG) Pair Selection

This greedy heuristic is adopted from an approximation algorithm for the weighted set covering problem [17]. This policy selects the content-site pair that has the largest overall maximum utility $\mu_{jk} > 0$ as follows:

$$\mu_{jk} = \frac{d^s_{jk}}{d^s_{jk} p^b_j + p^s_j} \tag{11}$$

The utility is defined as the amount of demands that can be served divided by the total cost incurred for serving requests on content k by site j. SCG has the benefit of optimizing rental cost and can be adopted in the second step of DPC.

Algorithm 3 The UAGSA Algorithm

```
1: $f(x^*) \leftarrow \infty$
2: for 1 to $I^{max}$ do
3:     Call Greedy Heuristic
4:     Update $u_j^t$ for all sites
5:     if $f(x) < f(x^*)$ then
6:         $x^* \leftarrow x$
7:     end if
8: end for
```

4.2 The UAGSA Algorithm

Based on SCG/SFG, we design UAGSA, which iterates I^{max} times to search for efficient solutions. In Algorithm 3, x denotes a feasible solution and $f(x)$ denotes the performance of the solution; I^{max} is a predetermined parameter and we will discuss how to choose I^{max} in Section 6. The best overall solution (denoted by x^*) is kept as the result. In each iteration it utilizes SFG/SCG to construct a feasible solution that can improve resource utilization in CDN sites (Line 3 of Algorithm 3). The time complexity of UAGSA is the same as Algorithm 1, i.e., $O(KMN)$. Based on SFG, SCG, and UAGSA, we develop two algorithms, SFG-UAGSA, and SCG-UAGSA. SFG-UAGSA is advantageous in maximizing demands under given resources and can be used in Step 1 of DPC. SCG-UAGSA is beneficial in minimizing rental cost and thus can be used in Step 2 of DPC.

We first use SFG-UAGSA as an example to introduce how UAGSA works. SFG-UAGSA runs a modified version of SFG to produce a feasible solution in each iteration t. For selecting pairs, SFG-UAGSA first selects a site with the largest α_j given by Equation (12), where u_j^t is defined as a utilization parameter for each site j in each iteration t to denote bandwidth/VM utilization at site j. Then SFG-UAGSA assigns the site with the content of which the total demands servable by the site is the largest among all contents.

$$\alpha_j = \gamma_j u_j^t \tag{12}$$

In the first iteration, u_j^t is initialized as 1 for each site. In each of the following iterations, after running the greedy heuristic, a feasible solution is produced and u_j^t will be updated for each site as follows:

$$u_j^t = u_j^{t-1} \frac{B_j}{B_j - B_j^r} \tag{13}$$

where $\frac{B_j - B_j^r}{B_j}$ is the utilization of bandwidth/VM capacities. Low utilization causes a decreasing α_j that gives a high priority to under-utilized sites, which can be better utilized in the next iteration. Hence in each iteration the priorities (α_js) of the sites are fixed based on u_j^ts, which are updated based on resource utilization in the latest feasible solution. This tends to produce solutions with efficient resource utilization. Since probably such solutions are also efficient in terms of maximizing supported demands, UAGSA can quickly and effectively find efficient solutions that can maximize demands supported by given resources. The best overall solution that maximizes supported demands is kept as the result.

Algorithm 4 The CRB Algorithm

```
1: Initialize $d_{ik}$ for each content $k$ and site $j$
2: $d_{ik}^r \leftarrow d_{ik} * r_{i0k}$ for each content $k$ and site $j$
3: $N' \leftarrow N$
4: for $j \leftarrow 1$ to $N$ do
5:     if site $j$ is overloaded then
6:         for $j \leftarrow 1$ to $M$ do
7:             for $k \leftarrow 1$ to $K$ do
8:                 if $\sum_{i \in M} \sum_{k \in K} d_{ik} r_{ijk} > B_j$ and $d_{ik} \neq 0$ then
9:                     $d \leftarrow \min(r_{ijk} d_{ik}, d_j - B_j)$
10:                    $r_{ijk} \leftarrow r_{ijk} - d/d_{ik}$
11:                    $r_{i0k} \leftarrow r_{i0k} + d/d_{ik}$
12:                    $d_{ik}^r \leftarrow d_{ik}^r + d$
13:                end if
14:            end for
15:        end for
16:    end if
17: end for
18: while true do
19:     Select a pair $(j, k)$ by modified SFG
20:     if no pair is found then
21:         Return
22:     else
23:         Call Replace$(j, k)$
24:     end if
25: end while
```

SCG-UAGSA works in a similar way to SFG-UAGSA. The only difference is that SCG-UAGSA runs a modified version of SCG to produce a feasible solution in each iteration t. That is, SCG-UAGSA selects the pair with the largest β_{jk}:

$$\beta_{jk} = u_j^t * \mu_{jk} \tag{14}$$

Equation (14) emphasizes on both cost efficiency (μ_{jk}) and resource utilization (u_j^t). Hence SCG-UAGSA may derive solutions that can minimize rental costs and efficiently utilize available resources. The best overall solution that minimizes rental costs is kept as the result.

According to our simulation results, we suggest using SFG or SFG-UAGSA for step 1. For step 2, we suggest using SCG or SCG-UAGSA. Iterative algorithms SFG-UAGSA and SCG-UAGSA can deliver better performance than their greedy competitors; however, the greedy heuristics are faster than the iterative algorithms. In this case, one can choose proper greedy or iterative heuristics according to performance and speed requirements in practical implementations.

5 The CRB Algorithm

In Section 4 we have presented long term provisioning and caching algorithms for renting cloud resources and building cloud-based CDNs. Since resource provisioning is comparatively less flexible, to timely deal with the dynamic demand patterns, in this section we further present a more flexible short term algorithm, referred to as the *Caching and Request Balancing* (CRB) algorithm. By leveraging the previous runs

Algorithm 5 Place (j, k)

```
1:  if content $k$ is not cached at site $j$ then
2:      if $S_j^r > 0$ then
3:          $S_j^r \leftarrow S_j^r - 1$
4:      else
5:          Among all contents cached at site $j$, select the content
            $k'$ that serves the least amount of demands
6:          if $d_{jk'}^s < d_{jk}^s$ then
7:              for each location $i$ in site $j$'s coverage do
8:                  $d_{ik'}^r \leftarrow d_{ik'}^r + r_{ijk'} d_{ik'}$
9:                  $r_{i0k'} \leftarrow r_{i0k'} + r_{ijk'}$
10:                 $r_{ijk'} \leftarrow 0$
11:             end for
12:             $B_j^r \leftarrow B_j^r + d_{jk'}^s$
13:             Remove content $k'$ from site $j$
14:         else
15:             $N' \leftarrow N' - j$
16:             Return
17:         end if
18:     end if
19:     Cache content $k$ at site $j$
20: end if
21: for each location $i$ in site $j$'s coverage do
22:     if $d_{ik} \neq 0$ then
23:         $d \leftarrow \min(d_{ik}^r, B_j^r)$
24:         $r_{ijk} \leftarrow r_{ijk} + d/d_{ik}$
25:         $r_{i0k} \leftarrow r_{i0k} - d/d_{ik}$
26:         $d_{ik}^r \leftarrow d_{ik}^r - d$
27:         $B_j^r \leftarrow B_j^r - d$
28:     end if
29: end for
```

of DPC, CRB dynamically adjusts the placement of contents and routing maps such that total demands supported by the previously leased resources can be maximized. Since no provisioning is involved and only dynamic adjustments are made, CRB is light-weight and thus can be more frequently executed than DPC. For instance, in Fig. 2 DPC is executed once to update provisioning and caching solutions for each interval Δt, while CRB is executed multiple times in each interval to adjust the DPC solution. By combining DPC and CRB, the update dilemma is naturally solved.

The CRB algorithm is shown in Algorithm 4. It dynamically "pull" and "push" demands among the root and sites to improve resource utilization. If requests arriving at a site is more than the predicted value, the site may not have enough resources to support all requests. Accordingly, the algorithm modifies the routing map to pull requests from overloaded sites to the root site (Lines 3 - 17 of Algorithm 4). If requests arriving at a site is less than the predicted value, the site can handle more demands. Thus, the algorithm pushes some requests from the root site to under-loaded sites (Lines 18 - 25 of Algorithm 4). The "push" operation works in a similar way to SCG/SFG described in Section 4, i.e., it works by repeatedly selecting content-site pairs and implementing the pairs. But it is different from SCG/SFG in two ways.

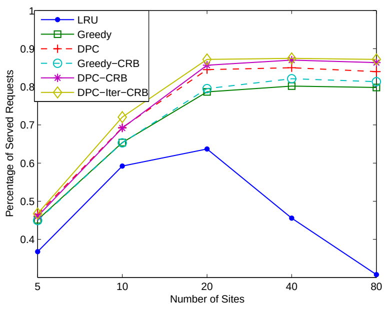
Firstly, for selecting pairs, we use a modified version of SFG to fit the scenario of request balancing. SFG selects a pair in two steps. It first selects a site with the largest γ_j and then assigns a content the site. However, since DPC tends to rent as many storages as used to save costs, probably no spare storage is left for request balancing ($S_j^r = 0$), rendering Equation(10) meaningless. Therefore, in the modified version, rather than selecting the site with the largest γ_j, the algorithm selects the site with the largest B_j^r. The intuition is that the sites with more under-utilized resources should be given higher priorities. Otherwise, such resources may be wasted.

Secondly, for implementing a pair, we design a placement procedure that places a content k at a site j, as shown in Algorithm 5. If site j has no spare storage, an unpopular content k' that covers the lowest amount of demands is removed from the site (Lines 5 - 13 of Algorithm 5). This is beneficial if content k is more popular than content k' at site j ($d_{jk'}^s < d_{jk}^s$), where $d_{jk'}^s$ is the amount of demands served by content k' at site j. Otherwise, the placement of content k at site j site cannot be favored and the procedure is terminated (Lines 15 and 16 of Algorithm 5). If content k has been successfully cached at site j, the routing map is then modified so that demands are migrated from the root to under-utilized sites (Lines 21 - 29 of Algorithm 5). The time complexity of Algorithm 4 is $O(KMN)$ due to the nested loops (Lines 4 - 7 of Algorithm 4).

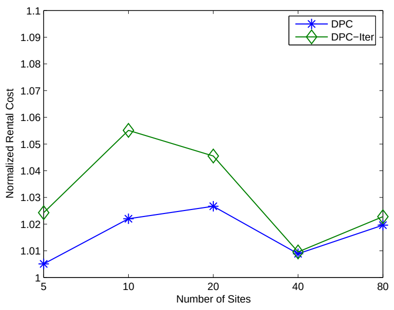
6 Performance Evaluation

In this section we present a performance evaluation study to assess the effectiveness of the proposed algorithms. We use Java to implement a simulator which generates a sequence of requests for a time period and serves demands over the period. It also has a scheduler which periodically runs the algorithms to perform provisioning and caching. In the simulation DPC is executed every 30 minutes and CRB is run at intervals of 10 minutes. In addition, the input predictions of demands for DPC and CRB are simply the historical demand data of the last 30 minutes and 10 minutes, respectively. Since both greedy heuristics and iterative heuristics can be applied in DPC, in the simulation results we use "DPC" to denote the basic version that adopts SFG and SCG, and we use "DPC-Iter" to denote the version using the iterative heuristics (SFG-UAGSA, and SCG-UAGSA). DPC-Iter iterates 10 times to yield a solution ($I^{max} = 10$). We then combine DPC and DPC-Iter with CRB to generate two algorithms: DPC-CRB and DPC-Iter-CRB.

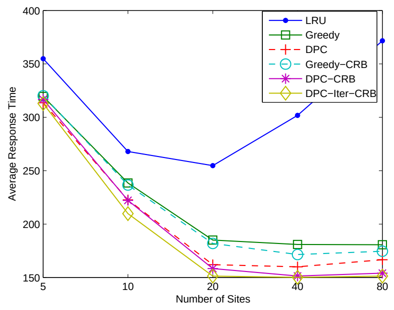
To understand the merits of our algorithms, we compare them with several baseline algorithms. The first baseline is called Greedy, which execute the set cover greedy heuristic to provision and cache contents, in the way that is described in Algorithm 1. In addition, to show the separate effects of DPC and CRB, we will show the results of DPC only (denoted as DPC), and the Greedy heuristic with CRB (denoted as Greedy-CRB). Notice that Greedy and Greedy-CRB are hourly executed as hourly resource rental is commonly supported in cloud systems [1]. Further, we also test LRU (Least Recently Used) for comparison. In LRU, each user is routed to the closest cloud site in terms of network delays. Upon a cache



(a) Percentage of served requests

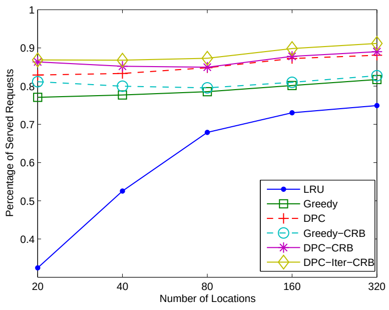


(b) Normalized rental cost

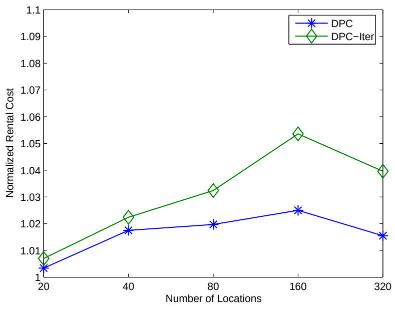


(c) Average response time

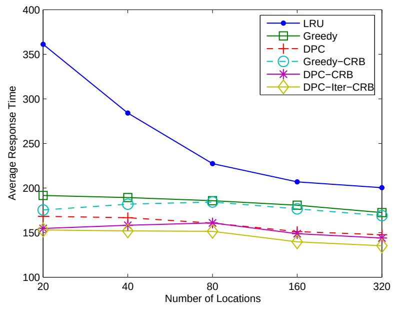
Fig. 3. Results with varying number of sites.



(a) Percentage of served requests



(b) Normalized rental cost



(c) Average response time

Fig. 4. Results with varying number of locations.

miss a cloud site retrieves the missed object from the root site and replaces the least recently used object with the missed object. Since LRU is only a caching policy but not a provisioning policy, when compared with our algorithms, we use the amount of resources provisioned in Greedy.

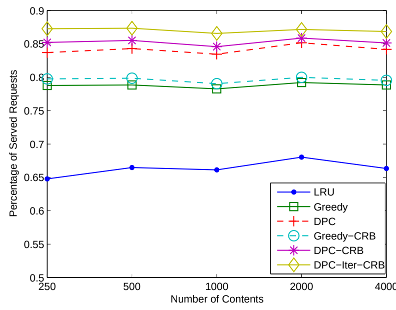
We are interested in 3 performance metrics. The first is the percentage of served requests, defined as the ratio of requests served by the cloud CDN (with QoS constraints satisfied) to the total number of requests. The second is the rental costs spent by the proposed algorithms. Hence, normalized rent cost is used as the second performance metric. It is defined as the total rental cost of an algorithm divide by that of the Greedy heuristic. Since CRB is irrelevant to resources provisioning, Greedy, Greedy-CRB, and LRU provision the same amount of resources with the same cost. Similarly, DPC and DPC-CRB spend the same cost. Therefore, in the following results (e.g., Fig. 3(b)) we only show the normalized rent costs of DPC and DPC-Iter. To characterize performance from users' perspective, we use average response time as the third metric.

In our experiments, the geo-distributed cloud sites and locations are randomly generated. We use the geographic distance between a site and a location as an indicator of delay. The round-trip delays (RTT) are emulated using manually injected delays in programs following the formula $RTT = 0.02 * Distance(km) + 5$ [5]. The target maximal average response delay is set as 150ms. With randomly generated sites and locations, we then use Q_{ij} to indicate whether the distance between a site and a location is qualified for content distribution. The leasing prices of cloud resources (i.e., bandwidth, VM, and storage) are also randomized by using the prices of Amazon EC2 and Nirvanix. The population p_i of each location i is uniformly distributed in the range $[100, 200]$. This parameter determines the request arrival rate d_i at each location according to $d_i = \frac{p_i \sum d_i}{\sum p_i}$. Accordingly, the number of locations is irrelevant to the total amount of incoming requests, but it can disperse the requests at each location. The number of locations can be covered by a site falling in the range $[2, 18]$. In fact, the choice of this parameter does not impact the performance of our algorithms (results not shown here).

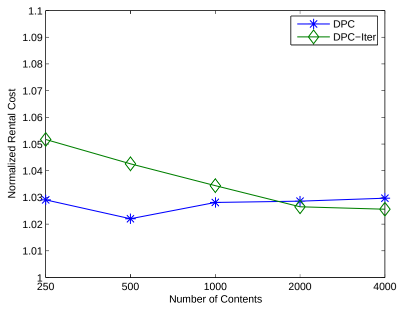
By default, we simulate a cloud across 20 cloud sites to serve requests from 80 locations. This setting represents the fact that users from many different locations are served by less cloud sites. We will also show the results of more sites and more locations. Firstly, we keep other settings fixed and only vary the number of sites in the cloud CDN. Fig. 3 depicts the corresponding results. Then, the number of locations is varied and the corresponding results are shown in Fig. 4.

The default number of content is set as 500. We also vary the number of contents in the range $[250, 4, 000]$ and the corresponding results are shown in Fig. 5, which confirms that the performance with a large number of contents is similar to that with a small number. Following prior works [18], [9], we assume that the popularity of contents is governed by a Zipf-Mandelbrot distribution with shape parameter α and plateau parameter q. By default, we let $\alpha = 1$ and let $q = 0.5$. We will also vary α in the range $[0.6, 1.4]$ and vary q in the range $[0, 2]$. Fig. 6 and Fig. 7 depict the corresponding results.

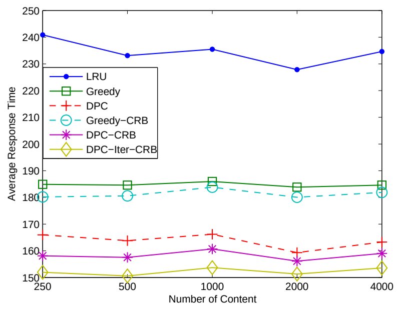
To model the dynamic variation of user demands, we let the arrival of requests conform to a non-homogeneous Poisson process, i.e., a Poisson process with rate parameter $\lambda(t)$. Following [19], we let the rate parameter λ be a piecewise linear function of time t (in minutes) with a linear rate δ_d, which is the increment of λ per second. We let $\lambda(t)$ vary in



(a) Percentage of served requests

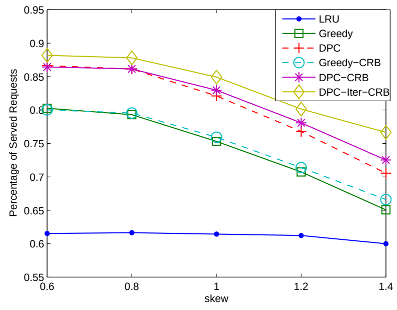


(b) Normalized rental cost

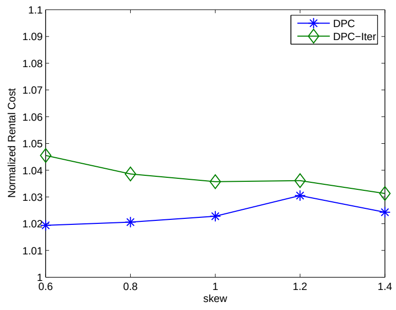


(c) Average response time

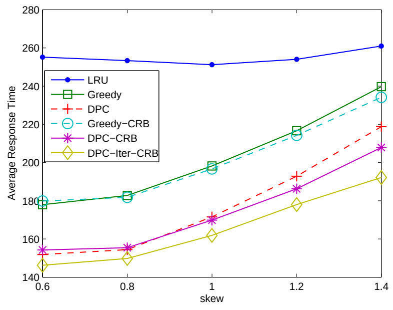
Fig. 5. Results with varying number of contents.



(a) Percentage of served requests



(b) Normalized rental cost



(c) Average response time

Fig. 6. Results with varying skew.

the range of $[200, 1200]$ according to the following function:

$$\lambda(t) = \begin{cases} 200 + \delta_d(t - nT), \ if \ nT \le t < (n+0.5)T \\ 1200 - \delta_d(t - nT), \ if \ (n+0.5)T \le t < (n+1)T \end{cases} \quad (15)$$

where $T = \frac{2000}{\delta_d}$ and $n = 1, 2, ...$ We then vary δ_d in the range $[0.5, 8]$ to study the influence of dynamically varying demands and show the dynamic resilience of our algorithms. The corresponding results are shown in Fig. 8.

The results in Figs. 3 - 8 show that DPC-CRB and DPC-Iter-CRB significantly outperform Greedy and LRU under various scenarios. Also, DPC-Iter-CRB outperforms others over all cases, indicating the effectiveness of UAGSA. The performance gains of DPC and Greedy-CRB over Greedy indicate that both DPC and CRB have contributions in enhancing performance. Fig. 8(a) shows when δ_d is small the performance gaps between DPC/DPC-CRB/DPC-Iter-CRB and Greedy/Greedy-CRB are comparatively small. As δ_d increases, the percentage of served requests drops for all algorithms and this trend is more severe for Greedy and Greedy-CRB. Hence, the performance gaps between DPC/DPC-CRB/DPC-Iter-CRB and Greedy/Greedy-CRB sharply increases. This suggests that DPC can effectively resist dynamism in incoming requests.

Also, the results in Figs. 3(b) - 8(b) show that our algorithms incur about 2%-5% more costs than Greedy. However, given the significant advantage of our algorithms in performance, it is worth spending such few additional costs in exchange of serving more requests. It may be noticed that the costs of DPC and DPC-Iter are normalized to Greedy and thus the plots in Figs. 3(b) - 8(b) do not reflect how the absolute costs really vary with varying parameters. in Further, it is shown in that DPC-Iter-CRB delivers the best performance over all cases but it also incurs more rental costs than other algorithms.

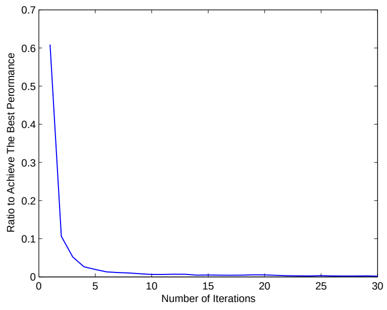
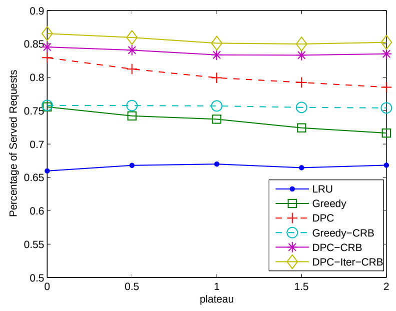


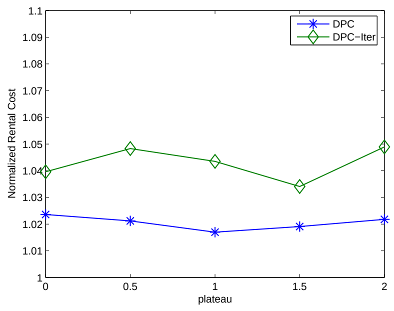
Fig. 9. The number of times that the best solution is available in the x-th iteration of UAGSA out of all experiments

Finally, Fig. 9 depicts how many times that the best solution is available in the x-th iteration out of the total number of experiments. The x axis is the number of iterations used in UAGSA and the y axis is the proportion of times that the x-th iteration obtains the best solution over the total number of experiments. In this experiment I^{max} is set as 30. It is shown that about 80% of the best solutions are obtained within 5 iterations. Indeed in 60% of all cases the best solution is achieved in the first iteration, i.e., UAGSA is useless in these cases. Lessons learnt from this experience seem to suggest that I^{max} can be set in the range [5, 20], which covers most cases that the best solution can be found in UAGSA.

7 Conclusions

This paper has addressed the resource provisioning and replica placement problems for cloud-based CDNs with an emphasis on handling dynamic demand patterns. We have proposed


(a) Percentage of served requests


(b) Normalized rental cost

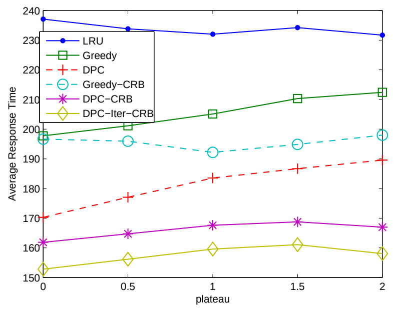
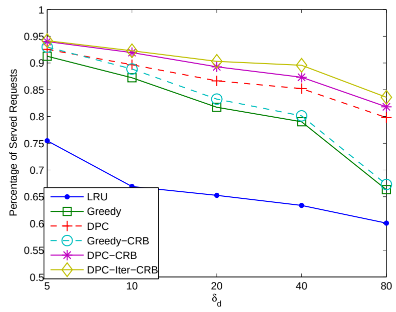
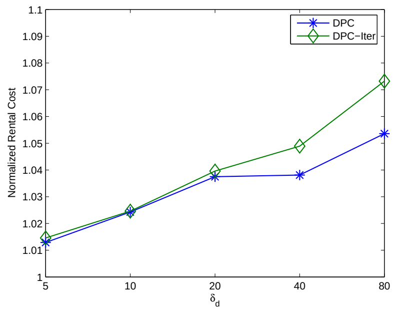

(c) Average response time

Fig. 7. Results with varying plateau.


(a) Percentage of served requests


(b) Normalized rental cost

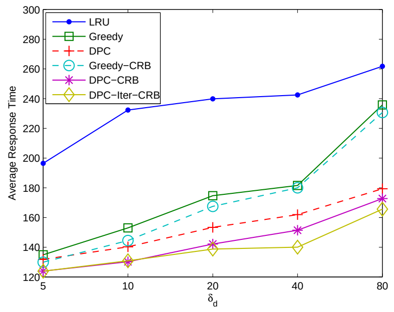

(c) Average response time

Fig. 8. Results with varying δ_d.

a set of novel algorithms to solve both the long-term provisioning and caching problem and the short-term caching and request balancing problem. Firstly, we have proposed the DPC algorithm, which rents cloud resources to build CDNs and caches contents on the cloud CDNs so that the total rental cost can be minimized while all demands are served. For each step of DPC, we have designed both greedy and iterative heuristics. Secondly, we have presented the CRB algorithm, which dynamically adjusts the placement of replicas and routing maps based on the previously run solutions of DPC such that total demands supported by the rented resources can be maximized at runtime. The simulation results have shown that our algorithms DPC-CRB and DPC-Iter-CRB constantly outperform Greedy and LRU under various scenarios. Also, it has been shown that DPC and CRB are adaptive to dynamic environments.

References

[1] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.
[2] "Four Reasons We Choose Amazon's Cloud as Our Computing Platform," The Netflix "Tech" Blog, December 2010.
[3] F. Chen, K. Guo, J. Lin, and T. La Porta, "Intra-cloud Lightning: Building CDNs in the Cloud," in *Proc. IEEE INFOCOM*, 2012.
[4] F. Wang, J. Liu, and M. Chen, "CALMS: Cloud-Assisted Live Media Streaming for Globalized Demands with Time/Region Diversities," in *Proc. IEEE INFOCOM*, 2012.
[5] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F.C.M. Lau, "Scaling Social Media Applications into Geo-Distributed Clouds," in *Proc. IEEE INFOCOM*, 2012.
[6] B. Li, M. Golin, G. Italiano, and X. Deng, "On the optimal placement of web servers in the Internet," in *Proc. IEEE Infocom*, 1999.
[7] P. Krishnan, D. Raz, and Y. Shavitt, "The Cache Location Problem," *IEEE/ACM Trans. Netw.*, vol. 8, no. 5, pp. 568-582, 2000.
[8] K. Kalpakis, K. Dasgupta, and O. Wolfson, "Optimal placement of replicas in trees with read, write, and storage costs," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 6, pp. 628-637, 2001.
[9] S. Borst, V. Gupta, and A. Walid, "Distributed Caching Algorithms for Content Distribution Networks," in *Proc. IEEE INFOCOM*, 2010.
[10] D. Niu, H. Xu, B. Li, and S. Zhao. "Quality-Assured Cloud Bandwidth Auto-Scaling for Video-on-Demand Applications," in *Proc. IEEE INFOCOM*, 2012.
[11] J. Dai, Z. Hu, B. Li, J. Liu, and B. Li. "Collaborative Hierarchical Caching with Dynamic Request Routing for Massive Content Distribution," in *Proc. IEEE INFOCOM*, 2012.
[12] Z. Shen, J. Luo, R. Zimmermann, and A.V. Vasilakos, "Peer-to-peer Media Streaming: Insights and New Developments," *Proc. IEEE*, vol. 99, no. 12, pp. 2089-2109, 2011.
[13] D. Applegate, A. Archer, V.G.S. Lee, and K. Ramakrishnan, "Optimal Content Placement for a Large-Scale VoD System," in *Proc. ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2010.
[14] G. Gursun, M. Crovella, and I. Matta, "Describing and Forecasting Video Access Patterns," in *Proc. IEEE INFOCOM Mini-Conference*, 2011.
[15] X. Tang and J. Xu, "QoS-Aware Replica Placement for Content Distribution," *IEEE Trans. Parallel Distributed Systems*, vol. 16, no. 10, pp. 921-932, 2005.
[16] G. Rodolakis, S. Siachalou, and L. Georgiadis, "Replicated Server Placement with QoS Constraints," *IEEE Trans. Parallel Distributed Systems*, vol. 17, no. 10, pp. 1151-162, 2006.
[17] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, pp. 233-235, 1979.
[18] M. Hefeeda and O. Saleh, "Traffic Modeling and Proportional Partial Caching for Peer-to-Peer Systems," *IEEE/ACM Trans. Netw.*, vol. 16, no. 6, pp. 1447-1460, 2008.
[19] W.A. Massey, G.A. Parker, and W. Whitt, "Estimating the parameters of a nonhomogeneous Poisson process with linear rate", *Telecommunication Systems*, vol. 5, no. 2, pp 361, 1996.
[20] R.M. Karp, "Reducibility among combinatorial problems," *50 Years of Integer Programming 1958-2008*, pp. 219-241, 2010.
[21] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, and P. Maechling, "Data Sharing Options for Scientific Workflows on Amazon EC2", in *Proc. 2010 ACM/IEEE Int'l Conf. High Performance Computing, Networking, Storage and Analysis*, 2010.

Menglan Hu received the B.E. degree from Huazhong University of Science and Technology, Wuhan, China, in 2007, and the Ph.D. degree from the National University of Singapore, Singapore, in 2012. He is currently a research fellow at the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests includes cloud computing, parallel and distributed systems, as well as scheduling and resource management.

Jun Luo received the B.S. and M.S. degrees in electrical engineering from Tsinghua University, Beijing, China, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the Swiss Federal Institute of Technology in Lausanne (EPFL), Lausanne, Switzerland, in 2006. From 2006 to 2008, he has worked as a Post-Doctoral Research Fellow with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. In 2008, he joined the faculty of the School of Computer Engineering, Nanyang Technological University, Singapore, where he is currently an Assistant Professor. His research interests include wireless networking, mobile and pervasive computing, distributed systems, multimedia protocols, network modeling and performance analysis, applied operations research, and network security. He is a member of the IEEE.

Yang Wang received the BS degree in applied mathematics from the Ocean University of China in 1989 and the MS and PhD degrees in computing science from Carleton University and the University of Alberta in 2001 and 2008, respectively. He is currently at IBM Center for Advanced Studies (CAS), Atlantic, University of New Brunswick, Fredericton, Canada. Before joining CAS Atlantic in 2012, he was a research fellow at the National University of Singapore from 2010 to 2012. Before that, he was a research associate at the University of Alberta, Canada, from August 2008 to March 2009. His research interests include scientific workflow computation and virtualization in Clouds and resource management algorithms.

Bharadwaj Veeravalli received his BSc in Physics, from Madurai-Kamaraj University, India in 1987, Master's in Electrical Communication Engineering from Indian Institute of Science, Bangalore, India in 1991 and PhD from Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India in 1994. He did his post-doctoral research in Concordia University, Montreal, Canada, in 1996. He is currently with the Department of Electrical and Computer Engineering at the National University of Singapore, Singapore, as a tenured Associate Professor. His research interests include Cloud/Grid/Cluster Computing, Scheduling in Parallel and Distributed Systems, Bioinformatics & Computational Biology, and Multimedia Computing. He is one of the earliest researchers in Divisible Load Theory (DLT). He had secured several externally funded projects and published over 120 papers in high-quality journals and conferences. He has co-authored three research monographs in the areas of PDS, Distributed Databases, and Networked Multimedia Systems, in 1996, 2003, and 2005, respectively. He is currently serving the Editorial Board of IEEE Transactions on SMC-A, Multimedia Tools & Applications (M-TAP) and Cluster Computing, as an Associate Editor. Until 2010 he had served as an AE for IEEE Transactions on Computers. More information can be found in http://cnl-ece.nus.edu.sg/elebv/. He is a senior member of the IEEE and the IEEE computer society.