On Server Provisioning for Cloud Gaming

Yusen Li[1] Yunhua Deng[2] Xueyan Tang[3] Wentong Cai[3] Xiaoguang Liu[1] Gang Wang[1]
[1]Nankai-Baidu Joint Lab, Nankai University, Tianjin, China
[2]Huawei Technologies, Shenzhen, China
[3]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[1]{liyusen, liuxg, wgzwp}@nbjl.nankai.edu.cn, [2]dengyunhua1@huawei.com, [3]{asxytang, aswtcai}@ntu.edu.sg

ABSTRACT

Cloud gaming has gained significant popularity recently due to many important benefits such as removal of device constraints, instant-on and cross-platform, etc. The properties of intensive resource demands and dynamic workloads make cloud gaming appropriate to be supported by an elastic cloud platform. Facing a large user population, a fundamental problem is how to provide satisfactory cloud gaming service at modest cost. We observe that software maintenance cost could be substantial compared to server running cost in cloud gaming. In this paper, we address the server provisioning problem for cloud gaming to optimize both server running cost and software maintenance cost. We find that the distribution of game softwares among servers triggers a trade-off between the software maintenance cost and server running cost. We formulate the problem with a stochastic model and employ queueing theories to conduct solid theoretical analysis. We then propose several classes of algorithms to approximate the optimal solution. The proposed algorithms are evaluated by extensive experiments using real-world parameters. The results show that the proposed algorithms are computationally efficient, nearly cost-optimal and highly robust to dynamic changes.

CCS CONCEPTS

- **Information systems → Multimedia information systems**;
- **Mathematics of computing → Combinatorial optimization**;
- **Computer systems organization → Cloud computing**;

KEYWORDS

Cloud Gaming; Service Cost; Server Provisioning; Software Distribution; Heuristic Algorithms.

1 INTRODUCTION

High-end video games such as World of Warcraft (WoW) can traditionally run only on powerful machines due to massive demands for computation, memory, and storage resources. Cloud gaming, which has gained significant popularity recently, has been adopted as a variable means to let players enjoy high-end video games on lightweight devices such as laptops, tablets and smart phones. The basic idea of cloud gaming is to run games on cloud servers while players interact with games through thin clients (see Figure 1). Specifically, cloud servers run the games, render and encode their graphical outputs into videos, and stream the videos to networked thin clients. The clients decode and display the video steams for players to interact with the games, and send control command inputs by players to cloud servers. In this way, the major computing load is transferred from the client side to the cloud and thus the device constraints for the players are removed. Moreover, cloud gaming also allows players to start games immediately without time-consuming software downloads and installations. Due to all these important benefits, cloud gaming has attracted a great deal of interests from both academia and industry [2, 4, 9, 21].

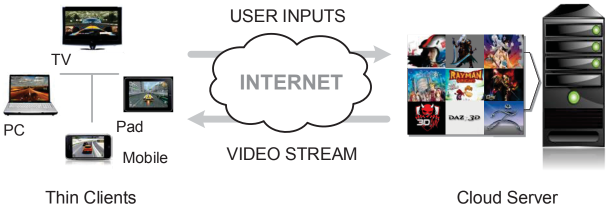

Figure 1: Cloud Gaming

Cloud gaming service has intensive resource demands and dynamic workloads. The elastic and on-demand nature of resource provisioning on public cloud infrastructures makes them attractive and suitable for supporting cloud gaming service [30]. When using public cloud infrastructures, the costs charged for the cloud resources used are the major operational expenses for cloud gaming service providers (CGSPs). A cloud gaming platform often provides services for hundreds of games. In order to run a game on a server, the game software must be installed on the server. Thus, in addition to the server running cost, CGSPs also need to pay for the software maintenance cost, which may include the storage cost etc. The software maintenance cost can be substantial compared to the server running cost. For example, the hourly cost of running a g2.2xlarge server (a popular server type for running graphic intensive applications such as games) in Amazon EC2 is $0.69 (for US East) [5]. The size of a high-end video game software is usually tens of GBs. The price of Amazon EBS storage is $0.1 per GB-month [5]. If hundreds of games are installed on a server, the hourly storage cost of the server can be as expensive as its running cost.

The software maintenance cost normally increases with the number of software copies that are installed. Due to the highly interactive nature of gaming, it is not desirable to load games from a shared repository in cloud gaming. This is because first, large network traffic may saturate the bandwidth of the shared repository, which could significantly slow down the game loading and

MM '17, October 23–27, 2017, Mountain View, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4906-2/17/10…$15.00
https://doi.org/10.1145/3123266.3123310

hurt user interactions; second, large scale games are usually I/O intensive, frequent read/write operations could lead to I/O conflicts and consistency issues if the game softwares are shared by multiple servers. Therefore, a practical way for cloud gaming is to install game softwares locally on each server [14]. The distribution of game softwares among servers triggers a trade-off between the software maintenance cost and server running cost. Installing fewer games on each server can reduce the software maintenance cost, but it will increase the number of servers needed to serve requests. A new server may have to be started for an incoming request even if there are running servers having spare computing resources available, because the requested game is not installed on these servers. This will raise the server running cost.

In this paper, we address the server provisioning problem for cloud gaming with the goal of minimizing the total server running cost and software maintenance cost. Existing studies on server provisioning for cloud gaming generally target at optimizing user interactivity and server running cost [19, 33]. So far, little effort has been devoted to exploring software maintenance cost and software distribution issues. Our work makes the first attempt to address this problem. The contributions of our study are summarized as follows:

First, we construct a stochastic model to characterize the cloud gaming system and study the complexity of the server provisioning problem. We propose an approach that organizes the games into disjoint groups and let each game group be served by a dedicated pool of servers. This approach greatly simplifies the system model without much loss in performance.

Second, by following practical cloud charging models, we formulate the optimization problem for game software distribution. We analyze the behavior of each server pool using queueing models, and employ Markov theory to derive the server running cost and software maintenance cost produced by each game group.

Third, we propose several categories of heuristic algorithms to divide the games into groups, which include simple algorithms, an ordered partition algorithm and a meta-heuristic algorithm. Extensive experiments are conducted using real-world parameters to evaluate the proposed algorithms and the results demonstrate the effectiveness of the algorithms.

The rest of this paper is structured as follows. The related work is summarized in Section 2. Section 3 presents the system model, problem formulation and service cost analysis. A set of heuristic algorithms are proposed in Section 4 to solve the server provisioning problem. In Section 5, the proposed algorithms are experimentally evaluated. Finally, conclusions are made and future work is discussed in Section 6.

2 RELATED WORK

Significant efforts have been devoted to cloud gaming in recent years. There have been several cloud gaming platforms from both industry and academia, such as Sony's PlayStation Now [4], NVIDIA's GeForce Now [2], Damai [1], GamingAnyWhere [21] and Rhizome [30]. Some research work has been conducted towards measuring and enhancing the performance of cloud gaming systems. The measurement work has concentrated on measuring the latency and network traffic of the existing commercial cloud gaming platforms [11, 25]. The enhancement work has focused on video encoding techniques and graphic rendering techniques for bit rate reduction [7, 17]. However, little effort has been made towards optimizing cloud resource provisioning in the above work.

There are some studies on selected resource management issues in cloud gaming. Hong et al. [19] considered how to consolidate game instances on the physical servers with the purpose of maximizing the service providers' profit while guaranteeing the players' Quality of Experience (QoE). Li et al. [26, 27] and Deng et al. [13] have studied the play request dispatching policies for minimizing the total service cost of a cloud gaming system using public cloud resources. Wu et al. [33] studied the request dispatching, server provisioning and video streaming bit rate settings jointly in a multi-region multi-datacenter cloud gaming system. The aim was to optimize the players' queueing delay, interaction latency as well as the system service cost. However, none of these studies considered the software maintenance cost and the game software distribution issue.

Server provisioning is a hot research topic in many fields. In large scale datacenters, server provisioning generally aims to improve resource utilizations and save the power or energy of machines [15, 28]. In the context of public clouds, server provisioning mainly addresses the problem of how to combine different instance (virtual machine) pricing models to serve time-varying demands at minimum monetary expense [20, 32]. For online gaming, the general purpose of server provisioning is to select servers from multiple geographically distributed datacenters for optimizing the interaction delay between users or minimizing server cost while meeting some performance requirements [23, 38]. However, the software maintenance cost was not considered in the above work.

Cloud gaming shares some similarities with the video-on-demand or live streaming applications [31, 34, 37]. However, in the latter applications, videos are often stored on one or a few networked storage devices which are shared by all the servers. In this setting, request dispatching among servers is trivial since the required data is always available no matter which server a request is assigned to. In contrast, in cloud gaming, each server maintains local copies of game softwares. This makes resource provisioning more challenging because the operation cost will also be influenced by request dispatching policies. The game software distribution problem is relevant to the data replication, data partition and data placement problems in many distributed systems [16, 22, 35, 36]. These problems usually concern how to distribute data among different locations for optimizing the consistency, access latency, and update cost of data. However, the storage cost has seldom been considered.

Our system model for server provisioning is also similar to multi-skill call center (MSCC) [10], where servers correspond to agents and games correspond to skills. The agents in the call center serve customers by answering calls. In a MSCC, calls have different types, which require different agent skills. Different agents may have different skill set, which lead to different hiring costs. A fundamental problem in a MSCC is how to allocate agents to meet the service level requirement (e.g., an acceptable call denial rate) while minimizing the hiring cost. However, there are some key differences between these two models: first, each agent can only serve one customer at a time in MSCC while a server can host multiple game instances in our model; second, the skill set of each agent is given

Table 1: Summary of Key Notations

Notation	Definition
N	the number of games
$\mathbf{G} = \{g_1, \ldots, g_N\}$	the set of games
$\lambda(g_i)$	the request arrival rate for game g_i
$1/\mu$	the mean session length of play requests
C	the capacity of each server
$\mathbb{P} = \{\mathbf{G}_1, \mathbf{G}_2, \ldots\}$	a partition of $\mathbf{G}$, where $\mathbf{G}_i$s are game groups
λ_i	the total request arrival rate for the games in $\mathbf{G}_i$
$U(\mathbf{G}_i)$	the total busy hours per unit time for all the servers of $\mathbf{G}_i$
c_s	the cost rate of running a server
$c_m(g_i)$	the cost rate of a software copy of g_i

in MSCC while the games to be installed on each server are to be determined in our model; finally, the number of games (often over hundreds) is much larger than the number of skills (generally less than 10), which makes our problem more challenging.

A very preliminary study on the game distribution problem in cloud gaming has been conducted by David et al. [14]. They first revealed the game distribution strategy used by the cloud gaming company OnLive and claimed that OnLive's approach incurs a huge waste of storage space. Then, they proposed a more efficient hill-climbing algorithm to reduce the storage space usage in game distribution. Our work differs from David et al.'s work in many aspects: (1) we consider both server running cost and software maintenance cost while only the storage space is considered in their work; (2) we focus on cloud gaming supported by public cloud infrastructures while they consider a private cluster; (3) we allow a server to run multiple game instances concurrently while a server can only run one game instance at a time in their model.

3 PROBLEM FORMULATION

3.1 System Model

We shall model the cloud gaming system as a stochastic process. Consider a cloud gaming platform which provides services for N different games. Denote by $\mathbf{G} = \{g_1, g_2, \ldots, g_N\}$ the set of N games. For each game $g_i \in \mathbf{G}$, we assume that the arrivals of play requests for g_i follow a Poisson process with an arrival rate $\lambda(g_i)$. Each play request corresponds to a *session* which is defined as the period when the requested game instance is running.

When a play request arrives, if there are some servers having sufficient residual capacity to handle the request and having the requested game installed, the request will be assigned to one of these servers according some specific dispatching policy. Otherwise, if no such server is available, a new server is started to accommodate the request. In general, once a game instance starts, it will run on the same server during the entire game session. Migrating game instances from one server to another is not preferable due to large migration overheads and interruption to game play. Suppose the cloud servers are homogeneous with the same capacity. Each server can run up to C game instances concurrently. We assume in this paper that there is an unlimited supply of servers from the cloud infrastructure, while the proposed analysis can easily be applied to the scenario where a fixed number of servers are specified.

The goal of server provisioning is to determine the set of games to be installed on each server so that the total service cost (namely, the sum of server running cost and software maintenance cost) is minimized. If games are allowed to be installed on servers arbitrarily, various servers could install different but overlapping sets of games. In this case, the system behaves as a complex queueing network which is very difficult to analyze. Moreover, the number of games and game popularities can change dynamically in practice, which requires server provisioning to be resilient to changes. Therefore, it is necessary to simplify the system model for designing simple, efficient and robust server provisioning strategies.

To simplify the system model, we consider partitioning the set of games $\mathbf{G}$ into disjoint subsets. Let $\mathbb{P} = \{\mathbf{G}_1, \mathbf{G}_2, \ldots\}$ denote a partition, where $\mathbf{G}_1, \mathbf{G}_2, \ldots$ are the subsets such that $\bigcup_{\mathbf{G}_i \in \mathbb{P}} \mathbf{G}_i = \mathbf{G}$ and $\mathbf{G}_i \cap \mathbf{G}_j = \emptyset$ for $i \neq j$. We call each subset $\mathbf{G}_i \in \mathbb{P}$ a *game group*. For each game group $\mathbf{G}_i$, we allocate dedicated cloud servers for serving the requests of the games in $\mathbf{G}_i$. All the games in $\mathbf{G}_i$ are installed on every server allocated to $\mathbf{G}_i$. The partitioning approach brings many benefits. First, after partition, the servers of each game group $\mathbf{G}_i$ are identical (with the same set of games installed) and independent of the servers of other game groups $\mathbf{G}_j$ ($j \neq i$). In this way, the overall system is decomposed into several independent queueing systems, one for each game group, which are easier to analyze. Second, once the partition is decided, the game software distribution is determined, which simply installs all the games in each group on every server of that group. Finally, even though games are partitioned into groups, the games in the same group will still share servers, which ensures high resource utilizations and flexibilities of servers.

3.2 Problem Formulation

By the partitioning approach, the server provisioning problem is transformed to finding the optimal partition of games that minimizes the total service cost. We say that a server is busy if it is serving at least one request. Under the "pay-as-you-go" billing model of today's clouds [5], both server running cost and software maintenance cost associated with a server are proportional to its busy hours. Servers can be switched off for saving costs when they are not busy. Given a partition $\mathbb{P}$, consider a game group $\mathbf{G}_i \in \mathbb{P}$. Let $U(\mathbf{G}_i)$ denote the total busy hours per unit time for all the servers of group $\mathbf{G}_i$. Let c_s denote the cost rate for running a server. For each game $g \in \mathbf{G}$, let $c_m(g)$ denote the cost rate for maintaining a software copy of g on a server. Then, the service cost per unit time of group $\mathbf{G}_i$ is given by

$$U(\mathbf{G}_i) \cdot (c_s + \sum_{g \in \mathbf{G}_i} c_m(g)). \tag{1}$$

The total service cost of partition $\mathbb{P}$ is the sum of the service cost of all groups, i.e.,

$$\sum_{\mathbf{G}_i \in \mathbb{P}} \big(U(\mathbf{G}_i) \cdot (c_s + \sum_{g \in \mathbf{G}_i} c_m(g))\big). \tag{2}$$

The game software distribution problem is to find the optimal partition $\mathbb{P}$ that minimizes the total service cost.

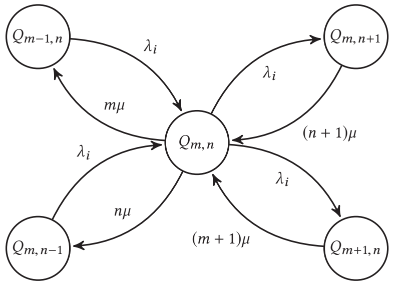

Figure 2: System state transition diagram of group G_i

3.3 Busy Hours Analysis

According to (2), in order to calculate the service cost produced by a given partition, the total busy hours $U(\mathbf{G}_i)$ should be analyzed for each game group $\mathbf{G}_i$. For each server in the system, the busy hours of the server are not only determined by the workload parameters (e.g., arrival rates, session lengths etc), but also highly dependent on the policies that assign play requests to the servers [26]. In this paper, we shall use the First Fit dispatching algorithm (the most commonly used algorithm for bin packing and job scheduling) as an example to illustrate the analysis of the busy hours for each server. The proposed analysis approach can be extended to other dispatching algorithms in a similar way.

Consider a game group $\mathbf{G}_i$. Let λ_i denote the total request arrival rate of all the games in $\mathbf{G}_i$, i.e., $\lambda_i = \sum_{g\in\mathbf{G}_i}(\lambda(g))$. For simplicity, we assume that the session lengths of requests follow an exponential distribution with a mean of $1/\mu$. The analysis can be extended to non-exponentially distributed session lengths using many approximation techniques [8, 18]. We label the servers of $\mathbf{G}_i$ as $s_1, s_2, ...$ in the order of their first starting times, a smaller index indicating earlier starting time. On each play request arrival, First Fit dispatches the request to the server with the smallest index among all servers which have sufficient residual capacity to host the game instance. If no such server is available, a new server is started to accommodate the request.

In order to analyze the busy probability of a server s_j ($j \geq 1$), consider the first j servers $s_1, s_2, ..., s_j$. Let $N(1, j-1)$ denote the number of requests being served by servers $s_1, s_2, \ldots, s_{j-1}$ and $N(j)$ denote the number of requests being served by server s_j. Let $Q(m, n)$ denote the system state where $N(1, j-1) = m$ and $N(j) = n$. Based on the above definitions, we build a Markov state diagram of system state transition, which is illustrated in Figure 2.

From state $Q(m, n)$, the system has a transition rate $m\mu$ to state $Q(m-1, n)$ and a transition rate $n\mu$ to state $Q(m, n-1)$. On the other hand, a new request arrives at the rate of λ_i. According to the First Fit rule, if the first $(j-1)$ servers are not fully occupied, the new request would be assigned to one of these servers. If these servers are fully occupied, the new request would be assigned to server s_j. Thus, from state $Q(m, n)$, the system has a transition rate λ_i to state $Q(m+1, n)$ if $m < (j-1)C$ (C is the server capacity), and a transition rate λ_i to state $Q(m, n+1)$ if $m = (j-1)C$ and $n < C$. Following these observations, the transition rates among all system states can be derived. Figure 3 shows the full state transition diagram (Markov chain) for the case of $j = 3$ and $C = 2$.

Let $p_{m,n}$ denote the probability that the system is in each state $Q(m, n)$. Let $J = (j-1)C$. According to the transition rates as shown in Figure 2, the balance (steady-state) equation of each $Q_{m,n}$ can be generated according to the following general formula:

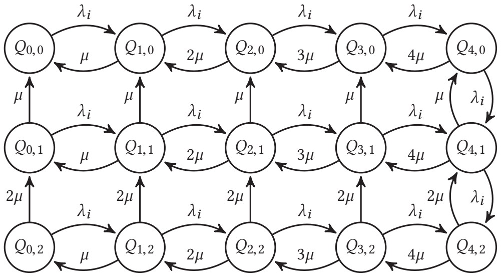

Figure 3: Markov chain for $j = 3, C = 2$

$$\big(\underbrace{m\mu}_{e_1} + \underbrace{n\mu}_{e_2} + \underbrace{\lambda_i}_{e_3} + \underbrace{\lambda_i}_{e_4}\big) \cdot p_{m,n} = \underbrace{\lambda_i \cdot p_{m-1,n}}_{e_5}$$
$$+ \underbrace{\lambda_i \cdot p_{m,n-1}}_{e_6} + \underbrace{(m+1)\mu \cdot p_{m+1,n}}_{e_7} + \underbrace{(n+1)\mu \cdot p_{m,n+1}}_{e_8},$$

where e_1 and e_5 hold for $m > 0$, e_3 and e_7 hold for $m < J$, e_2 holds for $n > 0$, e_4 holds for $n < C$ and $m = J$, e_6 holds for $n > 0$ and $m = J$, and e_8 holds for $n < C$.

Define matrix $P = [p_{0,0}\ p_{0,1}\ \cdots\ p_{J,C}]^T$ (the size of P is $(C+1)(J+1)$). The set of balance equations can be rewritten as

$$\Lambda P = 0, \tag{3}$$

where Λ is the matrix of coefficients. According to Markov theory, we also have

$$\sum_{m=0}^{J}\sum_{n=0}^{C} p_{m,n} = 1. \tag{4}$$

Thus, $p_{m,n}$s can be derived by solving the equations (3) and (4). It is worth noting that the equations could be solved efficiently since the matrix Λ is rather sparse.

Denote by $u(s_j)$ the busy probability of server s_j. It follows that

$$u(s_j) = p\{N(j) > 0\} = 1 - \sum_{i=0}^{J} p_{i,0}. \tag{5}$$

For the special case where $j = 1$, the behavior of the first server (i.e., s_1) can be modeled as an M/M/C/C queue. According to Erlang's loss Formula [6], it follows that

$$u(s_1) = p\{N(1) > 0\} = 1 - \frac{1}{\sum_{n=0}^{C}\frac{(\lambda_i/\mu)^n}{n!}}. \tag{6}$$

According to (5) and (6), the total busy hours per unit time for all the servers of group $\mathbf{G}_i$, i.e., $U(\mathbf{G}_i)$, can be calculated according to

$$U(\mathbf{G}_i) = \sum_{j=1}^{+\infty} u(s_j). \tag{7}$$

Note that $u(s_j)$ decreases rapidly as j increases. When j is sufficiently large, $u(j)$ will be very small and can be neglected in the practical calculations. Therefore, given a sufficient small threshold ε, $U(\mathbf{G}_i)$ can be approximated by $\sum_{j=1}^{j^*} u(s_j)$, where j^* is the maximum j such that $u(s_j) > \varepsilon$.

4 ALGORITHM DESIGN AND ANALYSIS

It can be shown that finding the optimal game partition to minimize the total cost is NP-hard in general. We develop several classes of partition heuristics which are computationally efficient.

4.1 Simple Algorithms

We first propose two simple partition algorithms:

1-Group Partition: In this approach, we put all the games into one group, i.e., let $\mathbb{P} = \{\{g_1, g_2, \ldots, g_N\}\}$. Each server installs all the games and thus servers can be shared by different games as much as possible, thereby minimizing the server running cost. However, the total number of software copies installed is very large and thus the software maintenance cost is high.

N-Groups Partition: This approach partitions the games into N groups, each with only one game, i.e., $\mathbb{P} = \{\{g_1\}, \{g_2\}, \ldots, \{g_N\}\}$. Thus, each server installs only one game. In this way, the number of software copies installed is small and thus the software maintenance cost is minimized. However, different games cannot share servers in this approach, which will increase the total server running cost.

4.2 Ordered Partition Algorithm

The basic idea of this algorithm is to sort all the games into a sequence according to their popularities (request rates), and then partition the ordered sequence into several segments, each of which corresponds to a game group. The motivation behind is to put the games with similar popularities (adjacent to each other in the ordered sequence) into the same group since the numbers of copies required by these games are similar. Without loss of generality, we assume that $\langle g_1, g_2, \ldots, g_N \rangle$ is an ordered sequence of all the games, where $\lambda(g_1) > \lambda(g_2) > \cdots > \lambda(g_N)$.

Given the ordered sequence $\langle g_1, g_2, \ldots, g_N \rangle$, the optimal partition that minimizes the total service cost among all the partitions of the sequence can be obtained by a dynamic programming algorithm. Let $\mathbb{P}^*(i)$ $(1 \le i \le N)$ denote the optimal partition when only the first i games in the sequence (i.e., $g_1, g_2, \ldots, g_i$) are considered. Let $cost(\mathbb{P}^*(i))$ denote the total service cost produced by partition $\mathbb{P}^*(i)$, which can be calculated according to (2). Let $\mathrm{G}[i, j]$ denote a game group composed by $g_i, g_{i+1}, \ldots, g_j$. Denote by $cost(\mathrm{G}[i, j])$ the service cost of group $\mathrm{G}[i, j]$, which can be calculated according to (1). It is easy to see that $\mathbb{P}^*(1) = \{g_1\}$ and $cost(\mathbb{P}^*(1)) = cost(\mathrm{G}[1, 1])$.

In order to find $\mathbb{P}^*(i)$, suppose the last game group of $\mathbb{P}^*(i)$ is $\mathrm{G}[i', i]$ $(1 \le i' \le i)$, it is easy to prove that $\mathbb{P}^*(i) = \mathbb{P}^*(i' - 1) \cup \{\mathrm{G}[i', i]\}$ and $cost(\mathbb{P}^*(i)) = cost(\mathbb{P}^*(i'-1)) + cost(\mathrm{G}[i', i])$. Therefore, for each $i > 1$, we have the following recurrence:

$$cost(\mathbb{P}^*(i)) = \min \begin{cases} cost(\mathbb{P}^*(i-1)) + cost(\mathrm{G}[i, i]) \\ cost(\mathbb{P}^*(i-2)) + cost(\mathrm{G}[i-1, i]) \\ \vdots \\ cost(\mathbb{P}^*(1)) + cost(\mathrm{G}[2, i]) \\ cost(\mathrm{G}[1, i]) \end{cases} \tag{8}$$

Based on (8), $cost(\mathbb{P}^*(N))$ can be calculated and the optimal partition $\mathbb{P}^*(N)$ can be derived accordingly. The pseudo code of the dynamic programming algorithm is shown in Algorithm 1 (referred to as *Ordered*). The variable $magic[j]$ denotes the size of the last game group (i.e., the right-most segment of the sequence) in the optimal partition when only the first j games (i.e., $g_1, g_2, \ldots, g_j$) are considered. So, the size of the last game group in $\mathbb{P}^*(N)$ is given by $magic[N]$. Similarly, $magic[N - magic[N]]$ denotes the size of the second last game group in $\mathbb{P}^*(N)$, and so on. Then, the size of every game group in $\mathbb{P}^*(N)$ can be derived and $\mathbb{P}^*(N)$ can be determined accordingly.

Algorithm 1 Ordered Partition Algorithm (*Ordered*)

```
$cost(\mathbb{P}^*(1)) := cost(\mathrm{G}[1, 1])$
$magic[1] := 1$
for each $i$ from 2 to $N$ do
    $k^* := \arg\min_{1 \le k \le i-1}(cost(\mathbb{P}^*(i-k)) + cost(\mathrm{G}[i-k+1, i]))$
    if $cost(\mathrm{G}[1, i]) < cost(\mathbb{P}^*(i-k^*)) + cost(\mathrm{G}[i, i-k^*+1])$ then
        $cost(\mathbb{P}^*(i)) := cost(\mathrm{G}[1, i])$
        $magic[i] := i$
    else
        $cost(\mathbb{P}^*(i)) := cost(\mathbb{P}^*(i-k^*)) + cost(\mathrm{G}[i, i-k^*+1])$
        $magic[i] := k^*$
    end if
end for
```

4.3 Genetic Algorithm

Genetic algorithms (GAs) have been shown very efficient in solving combinational optimization problems [12]. In this section, we apply the GA to our game partitioning problem. As illustrated in Section 4.2, given a sequence of the games, the optimal partition of the sequence that minimizes the service cost can be calculated by dynamic programming. Based on this fact, our GA aims to search for the best sequence of the games (i.e., the sequence whose optimal partition gives the minimum service cost). Then, the optimal partition of the games can be derived accordingly, which is the optimal partition of the best sequence.

We first illustrate the design of the GA for our problem:

Coding: The coding scheme in a GA determines how a candidate solution is represented by a chromosome-like data structure. For our problem, a solution refers to a sequence of the games. We adopt a simple approach which encode a sequence of games as a string composed by the indices of the games. For example, the sequence $\langle g_1, g_3, g_2, g_4, g_5 \rangle$ will be encoded as "1 3 2 4 5".

Selection: Selection is an operation to select two parent chromosomes for generating a new chromosome. We adopt a widely used selection approach proposed in [29]. Let K be the population size in each generation. We denote the K chromosomes in the t-th generation by $\mathrm{X^t} = (x_1^t, x_2^t, \ldots, x_K^t)$. Recall that each chromosome x_i^t represents a sequence of games. Denote by $\mathbb{P}(x_i^t)$ the optimal partition of the sequence represented by x_i^t. Let $cost(x_i^t)$ denote the service cost produced by $\mathbb{P}(x_i^t)$, i.e., $cost(x_i^t) = cost(\mathbb{P}(x_i^t))$, which can be calculated according to (8). Then, x_i^t will be selected as a parent according to the selection probability

$$p(x_i^t) = \frac{[cost_K(\mathrm{X^t}) - cost(x_i^t)]^2}{\sum_{x_i^t \in \mathrm{X^t}} [cost_K(\mathrm{X^t}) - cost(x_i^t)]^2},$$

where $cost_K(\mathrm{X^t}) = max\{cost(x_i^t) | x_i^t \in \mathrm{X^t}\}$, i.e., $cost_K(\mathrm{X^t})$ is the maximum cost among all solutions.

Crossover: Crossover is an operation to generate a new chromosome (i.e., child) from two parent chromosomes. In this paper, we consider two commonly used crossover operators:

(1) *One-point crossover.* In this crossover operation, one point (a position of the sequence) is randomly selected for dividing one parent. The set of games on one side (each side is chosen with the same probability) is inherited from one parent to the child, and the other games are placed in the order of their appearance in the other parent.

(2) *Two-point crossover.* In this crossover operation, two points are randomly selected for dividing one parent. There are two ways to generate the child based on the points selected: (i) The games outside the selected points are inherited from one parent to the child, and the other games are placed in the order of their appearance in the other parent; or (ii) The games between two points are inherited from one parent to the child, and the other games are placed in the order of their appearance in the other parent.

Mutation: Mutation is an operation to change the order of games in each chromosome generated by a crossover operator. A mutation operation can be viewed as a transition from a current solution to its neighborhood solution in local search algorithms. In this paper, we examine the following three mutation operators:

(1) *Shift.* In this mutation, a game at one position is removed and put at another position in the sequence. The two positions are randomly selected.

(2) *Swap.* In this mutation, the games at two different positions are exchanged. The two games to be swapped are randomly selected.

(3) *Mixed.* This mutation is a combination of one shift mutation and one swap mutation.

Based on the above definitions, let P^t and C^t be the parents and child chromosomes at generation t. The outline of the genetic algorithm is shown in Algorithm 2 (referred to as *Genetic*).

Algorithm 2 Genetic Algorithm (*Genetic*)

```
$t := 0$
Generate the initial population $\mathrm{P}^t$
Evaluate each chromosome in $\mathrm{P}^t$
while $t < Max\_Iterations$ do
    Select $K$ pairs of parents from $\mathrm{P}^t$
    Apply one of the above crossover operators to each of the selected pairs to generate $\mathrm{C}^t$
    Apply one of the above mutation operators to each of the chromosome in $\mathrm{C}^t$
    Select the top $K$ best chromosomes (the chromosomes with the smallest service costs) from $\mathrm{P}^t$ and $\mathrm{C}^t$ as $\mathrm{P}^{t+1}$
    $t := t + 1$
end while
```

Particularly, in the real implementation, we include a special chromosome which is generated using the ordered sequence of the games (as discussed in Section 4.2) in the initial population. In this way, the game partition obtained by *Genetic* will be always superior than the game partition obtained by *Ordered*. Note that, the partitions obtained by *Ordered* and *Genetic* are the optimal partitions of some specific game sequences. Therefore, *Ordered* and *Genetic* will always outperform *1-Group* and *N-Group* since the partitions given by *1-Group* and *N-Group* can be considered as the possible partitions of any game sequence.

5 EXPERIMENTS

We develop a discrete event driven simulator and conduct extensive experiments to evaluate the performance of the proposed algorithms. We first briefly introduce the simulation settings, and then present the evaluation results.

5.1 Simulation Settings

The models and parameter settings used in the simulation are based on measurements in a real cloud gaming system [14]. We simulate $N = 100$ games in the experiments. Each game g_i $(1 \leq i \leq 100)$ has a popularity and a game size, which are denoted by $p(g_i)$ and $s(g_i)$ respectively. According to [14], the popularities of games in cloud gaming follow Zipf's law. Thus, we let $p(g_i) = 1/i^{\alpha}$, where α is the shape parameter of the Zipf distribution function. In this way, a game with a smaller index has higher popularity, i.e., we have $p(g_1) > p(g_2) > \cdots > p(g_N)$. We collect the sizes of 100 randomly selected XBox games (*http://www.xbox.com/zh-CN/*) and use them as the game sizes in our simulations. For each game g_i, play request arrivals follow a Poisson process with an arrival rate of $\lambda(\mathrm{G}) \cdot p(g_i)/\sum_{g_i \in \mathrm{G}} p(g_i)$, where $\lambda(\mathrm{G})$ is the total arrival rate of all the games. On request arrivals, First Fit is used for dispatching the requests to the servers. The session length of each game request is randomly generated from an exponential distribution with a mean $1/\mu$.

We set the server running cost at c_s = \$0.69 per hour, which is the real cost of a g2.2xlarge instance of Amazon EC2 in Virginia. In practice, the software maintenance cost may include expenses for storage, license fees, etc. For simplicity, we take the storage cost as the software maintenance cost in the simulation. Let c_r denote the storage cost rate. We set c_r to \$0.1 per GB-month by default, according to the price of Amazon's EBS. Therefore, the software maintenance cost of a single copy of g_i is given by $c_m(g_i) = c_r \cdot s(g_i)$ ($s(g_i)$ is the size of game g_i). For *Genetic*, by testing various parameter specifications, we found that $K = 50$ (the population size) and $Max\_Iterations = 10000$ (the maximum number of iterations) work well. In the experiments, various parameter settings are tested. For each parameter setting, we calculate the partition generated by each algorithm and simulate a cloud gaming system for a period of 30 days to collect the service cost produced by the partition. By default, the server capacity (C) is set at 4 requests per server and the shape parameter α of Zipf's law is set at 2.0.

For comparison, we derive an approximate lower bound of the total service cost. Recall that *1-Group* installs all games on each server, which maximizes server utilizations. Therefore, it is reasonable to assume that the server running cost produced by *1-Group* is a lower bound of server running cost. On the other hand, *N-Group* makes each server serve only one game, which minimizes the number of software copies installed for each game. Thus, it is rational to assume that the software maintenance cost produced by *N-Group* is a lower bound of software maintenance cost. Therefore, a lower bound of the total service cost can be computed as the sum of *1-Group*'s server running cost and *N-Group*'s software maintenance cost. We normalize the total service cost produced by

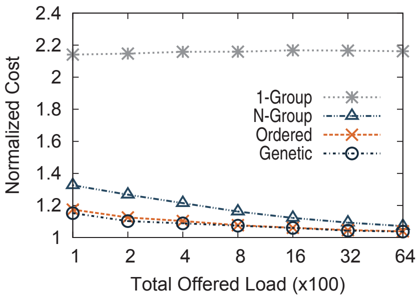

Figure 4: Impact of workload

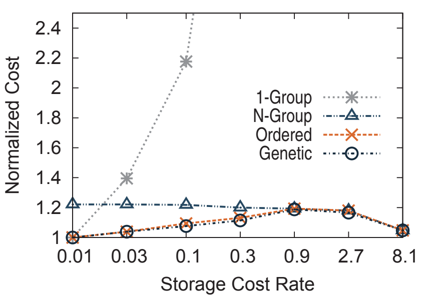

Figure 5: Impact of software maintenance cost rate

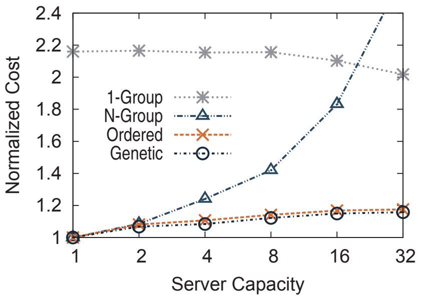

Figure 6: Impact of server capacity

each algorithm with respect to this lower bound, which we call the *normalized cost*. It is worth noting that the above lower bound is not limited to our proposed partitioning approach, but holds for any game distribution strategy. It is not necessarily achievable by any algorithm.

5.2 Performance under Different Parameter Settings

We first evaluate the impact of the total workload, which is defined by $\lambda(\mathbf{G})/\mu$. Figure 4 shows the normalized cost produced by each algorithm against different workload. Basically, the normalized cost (except for *1-Group*) decreases with increasing workload for all the algorithms. This is because when the request rate is high, all the servers have high utilizations for serving the continuously arriving requests in each algorithm, and thus, the server running costs produced by the algorithms are similar and are close to the optimal server running cost. Among all the algorithms, *Genetic* gives the best performance while *Ordered* is also very competitive. They both outperform the other two algorithms.

It also can be observed from Figure 4 that the normalized costs produced by *Genetic* and *Ordered* are very low in most cases, implying that the service costs produced by these two algorithms are close to the lower bound. Recall that the lower bound is the minimum service cost that can be produced by any game distribution strategy. It implies that our proposed partitioning approach is very effective. We also see that the normalized cost produced by *1-Group* is far higher than other algorithms. This is because all the servers install all the games in *1-Group*, which incurs high software maintenance cost. It confirms the motivation of our work that the benefit of optimizing software maintenance cost could be significant. In the rest experiments, unless otherwise stated, $\lambda(\mathbf{G})/\mu$ is set at 400 by default.

The results in Figure 5 illustrate how the ratio of software maintenance cost to server running cost influences the performance of the algorithms. We fix the server running cost rate c_s and vary the storage cost rate c_r in $[0.01, 0.03, 0.1, 0.3, 0.9, 2.7, 8.1]$. We see that all the algorithms (except for *N-Group*) produce similar costs when the storage cost rate is very small. This is because the software maintenance cost is very low in this case, which makes all the algorithms (except for *N-Group*) put all the games in one game group and thus produce similar costs. We also see that the performance of all the algorithms (except for *1-Group*) is comparable when the storage cost rate is large. In this case, the server running cost becomes insignificant compared to the software maintenance cost, implying that reducing the number of software copies is more important than grouping games together. All the algorithms (except for *1-Group*) generate similar partitioning results (many small game groups) and thus the costs produced are similar. In contrast, *1-Group* installs all the games on each server, which incurs much higher software maintenance cost compared to other algorithms.

Next, we examine how the server capacity influences the performance of the algorithms. Figure 6 shows the normalized cost produced by each algorithm as the server capacity varies from 1 to 32 (the GPU technology is currently able to support up to 32 game instances on a single board [3]). We see that the normalized cost produced by *N-Group* grows fast when the server capacity increases. This is because different games cannot share servers in *N-Group*. When the server capacity is large, most of the capacities are not utilized and thus the total server running cost is high. By contrast, servers are shared by all the games in *1-Group*, giving the best server utilizations compared to other algorithms. Therefore, as the server capacity grows, a trend of cost decrease is observed for *1-Group* in Figure 6. Similar to the previous results, *Ordered* and *Genetic* outperform *1-Group* and *N-Group* significantly in most of the cases.

Finally, we evaluate the impact of game popularity distribution on the performance. Figure 7 shows the normalized cost produced by each algorithm as the Zipf function's shape parameter α varies from 0.5 to 3. A small α will make the popularity distribution more uniform while a large α will make the popularity distribution more skewed. We see that the performance of *N-Group*, *Ordered* and *Genetic* are comparable when α is very large. This is because when α is very large, the game popularities are highly skewed, where a smaller number of games have very high popularities and dominate the cost. It is similar to the case of partitioning a small number of games each with a high request rate. According to the previous observations (in Figure 4), all the servers have high utilizations in this case and thus the costs produced by these three algorithms are similar. However, the software maintenance cost produced by *1-Group* is far higher than other algorithms when α is large since all the games are installed on each server in *1-Group*.

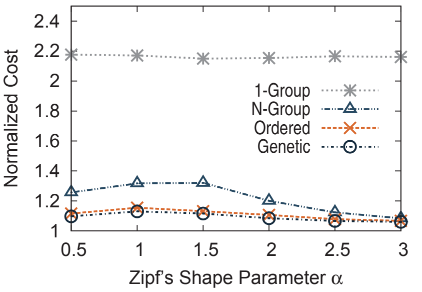

Figure 7: Impact of game popularity distribution

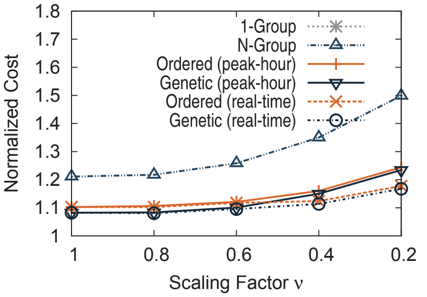

Figure 8: Robustness to workload dynamics

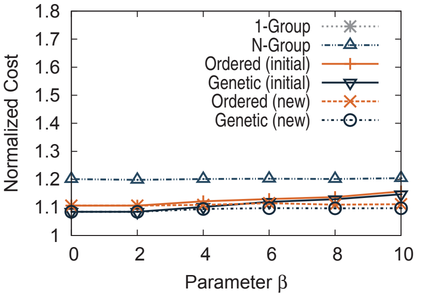

Figure 9: Robustness to popularity variations

5.3 Resilience to Dynamic Changes

As we know, the workload of online gaming is often dynamic. The workload variations in online gaming usually present the following two patterns [24]: (1) the number of players varies greatly in the period of a day, late evening is usually the peak hours while early morning is the slack period; (2) the numbers of players in different days are similar, implying that game popularities vary slowly at a large time scale. In this section, we study whether our proposed approaches are robust to these workload variations.

The experiments are designed as follows. Let ρ_{peak} be the workload (i.e., $\lambda(\mathbf{G})/\mu$) in peak hours. We first run the proposed algorithms to generate partitions according to ρ_{peak}, which we call *peak-hour partitions*. In order to examine how the *peak-hour partitions* perform in non-peak hours, we generate various workload for non-peak hours according to $\rho' = \nu \cdot \rho_{peak}$, where $0 < \nu < 1$ is a scaling factor. For each ρ', we run the proposed algorithms again to calculate partitions according to ρ', which we call *real-time partitions* for workload ρ'. It is easy to see that the *peak-hour partition* and *real-time partition* are the same for both *1-Group* and *N-Group*. After that, for each ρ', we run simulations with ρ' as the workload for all *peak-hour partitions* and *real-time partitions*.

Figure 8 shows the normalized cost produced by different partitions for different scaling factors ν when $\rho_{peak} = 400$. It can be seen that *peak-hour partitions* of *Ordered* and *Genetic* produce comparable performance to the corresponding *real-time partitions* in all the cases, and they outperform *1-Group* and *N-Group* significantly (*1-Group*'s normalized cost is higher than 2.0 and thus cannot be seen in Figure 8). It implies that *Ordered* and *Genetic* are very robust to workload variations. The partitions generated by *Ordered* and *Genetic* according to the workload in peak hours are also effective for non-peak hours. This is possibly because although the total workload varies greatly, the relative ranking of games by arrival rates keeps almost unchanged, giving rise to similar partition results for different workloads.

Next, we evaluate how the proposed algorithms perform when the game popularity changes. Given the initial game popularities, we first run algorithms to generate partitions which we call *initial partitions*. To simulate game popularity changes, we exchange the popularities between games (the intention here is to keep the Zipf's distribution) according to parameter β, an integer that defines the exchanging range. For each game g_i, we exchange g_i's popularity with a randomly selected game in the range $[g_{\min\{1,i-\beta\}}, g_{\max\{N,i+\beta\}}]$. It is easy to see that a larger β will lead to more significant popularity changes. After that, we run the proposed algorithms again to calculate the partitions according to the new game popularities, which we call *new partitions*. We then run simulations using the workload of new game popularities to evaluate all *initial partitions* and *new partitions*.

Figure 9 shows the normalized cost produced by different partitions for different β values. For *Ordered* and *Genetic*, we observe that the performance of the *initial partitions* of *Ordered* and *Genetic* are very close to the *new partitions* for small β values, which outperforms *1-Group* and *N-Group* significantly (again, *1-Group* cannot be seen since its normalized cost is higher than 2.0). Since game popularities change slowly (i.e., β should be small) in real world, it implies that *Ordered* and *Genetic* can perform well without frequent re-executions.

6 CONCLUSIONS

In this paper, we have investigated the server provisioning problem for optimizing the service cost in cloud gaming, taking both the software maintenance cost and game software distribution into account. We model the problem as a stochastic optimization problem and propose an effective simplification of the model. Several classes of computationally efficient heuristic algorithms are proposed, which are experimentally evaluated by simulations with real-world parameters. The results show that *Ordered* and *Genetic* perform quite well in most cases, and the partitioning approach is robust to dynamic changes. As a first attempt to the problem, we have assumed that some parameters of the system are homogeneous, such as the service level requirement, server capacity and the distribution of session lengths. A future direction is to study the impact of heterogeneities of these parameters.

7 ACKNOWLEDGMENTS

This work is partially supported by NSF of China (grant numbers: 61373018, 61602266 and 11550110491) and NSF of Tianjin (grant numbers: 16JCYBJC41900 and 17JCYBJC15300). This research is also supported by the National Research Foundation, Prime Minister's Office, Singapore under its IDM Futures Funding Initiative, and by Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2013-T2-2-067.

REFERENCES

[1] 2016. Damai. (2016). http://www.pyou.com/

[2] 2016. GeForce Now. (2016). http://www.geforce.com/

[3] 2016. NVIDIA GRID. (2016). http://www.nvidia.com/object/grid-technology.html

[4] 2016. PlayStation Now. (2016). https://www.playstation.com/

[5] 2017. Amazon EC2. (2017). https://aws.amazon.com/ec2/

[6] Ivo Adan and Jacques Resing. 2002. Queueing theory. (2002).

[7] Hamed Ahmadi, Saman Zad Tootaghaj, Mahmoud Reza Hashemi, and Shervin Shirmohammadi. 2014. A game attention model for efficient bit rate allocation in cloud gaming. *Multimedia Systems* (2014), 1–17.

[8] Onno J Boxma, JW Cohen, and N Huffels. 1979. Approximations of the mean waiting time in an M/G/s queueing system. *Operations Research* 27, 6 (1979), 1115–1127.

[9] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor C. M. Leung, and Cheng-Hsin Hsu. 2016. The Future of Cloud Gaming. *Proceedings of IEEE* 104, 4 (April 2016), 687–691.

[10] Mehmet Tolga Cezik and Pierre L'Ecuyer. 2008. Staffing multiskill call centers via linear programming and simulation. *Management Science* 54, 2 (2008), 310–323.

[11] Mark Claypool, David Finkel, Alexander Grant, and Michael Solano. 2014. On the performance of OnLive thin client games. *Multimedia Systems* (2014), 1–14.

[12] Lawrence Davis. 1991. Handbook of Genetic Algorithms. (1991).

[13] Yunhua Deng, Yusen Li, Xueyan Tang, and Wentong Cai. 2016. Server Allocation for Multiplayer Cloud Gaming. In *Proceedings of the 2016 ACM International Conference on Multimedia*. ACM, 918–927.

[14] David Finkel, Mark Claypool, Sam Jaffe, Thinh Nguyen, and Brendan Stephen. 2014. Assignment of games to servers in the OnLive cloud game system. In *2014 13th Annual Workshop on Network and Systems Support for Games*. IEEE, 1–3.

[15] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A Kozuch. 2010. Optimality analysis of energy-performance trade-off for server farm management. *Performance Evaluation* 67, 11 (2010), 1155–1171.

[16] Sushant Goel and Rajkumar Buyya. 2006. Data replication strategies in wide area distributed systems. *Enterprise Service Computing: from Concept to Deployment* 17 (2006).

[17] Mahdi Hemmati, Abbas Javadtalab, Ali Asghar Nazari Shirehjini, Shervin Shirmohammadi, and Tarik Arici. 2013. Game as video: bit rate reduction through adaptive object encoding. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 7–12.

[18] Per Hokstad. 1978. Approximations for the M/G/m queue. *Operations Research* 26, 3 (1978), 510–523.

[19] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. 2014. Placing virtual machines to optimize cloud gaming experience. *IEEE Transactions on Cloud Computing* (2014).

[20] Yu-Ju Hong, Jiachen Xue, and Mithuna Thottethodi. 2011. Dynamic server provisioning to minimize cost in an IaaS cloud. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 147–148.

[21] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: an open cloud gaming system. In *Proceedings of the 4th ACM Multimedia Systems Conference*. 36–47.

[22] Lei Jiao, Jun Lit, Wei Du, and Xiaoming Fu. 2014. Multi-objective data placement for multi-cloud socially aware services. In *2014 IEEE Conference on Computer Communications*. IEEE, 28–36.

[23] Kang-Won Lee, Bong-Jun Ko, and Seraphin Calo. 2005. Adaptive server selection for large scale interactive online games. *Computer Networks* 49, 1 (2005), 84–102.

[24] Yeng-Ting Lee, Kuan-Ta Chen, Yun-Maw Cheng, and Chin-Laung Lei. 2011. World of Warcraft avatar history dataset. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*. ACM, 123–128.

[25] Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. 2012. Are all games equally cloud-gaming-friendly? an electromyographic approach. In *11th Annual Workshop on Network and Systems Support for Games*. IEEE, 1–6.

[26] Yusen Li, Xueyan Tang, and Wentong Cai. 2014. On dynamic bin packing for resource allocation in the cloud. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. 2–11.

[27] Yusen Li, Xueyan Tang, and Wentong Cai. 2015. Play request dispatching for efficient virtual machine usage in cloud gaming. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 2052–2063.

[28] Minghong Lin, Adam Wierman, Lachlan LH Andrew, and Eno Thereska. 2013. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Transactions on Networking* 21, 5 (2013), 1378–1391.

[29] Tadahiko Murata, Hisao Ishibuchi, and Hideo Tanaka. 1996. Genetic algorithms for flowshop scheduling problems. *Computers and Industrial Engineering* 30, 4 (1996), 1061–1071.

[30] Ryan Shea, Di Fu, and Jiangchuan Liu. 2015. Rhizome: utilizing the public cloud to provide 3D gaming infrastructure. In *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 97–100.

[31] Feng Wang, Jiangchuan Liu, and Minghua Chen. 2012. CALMS: Cloud-assisted live media streaming for globalized demands with time/region diversities. In *IEEE Conference on Computer Communications 2012*. IEEE, 199–207.

[32] Wei Wang, Baochun Li, and Ben Liang. 2013. To reserve or not to reserve: optimal online multi-instance acquisition in IaaS clouds. In *Proceedings of the 10th International Conference on Autonomic Computing*. 13–22.

[33] Di Wu, Xue Zheng, and Jian He. 2014. iCloudAccess: cost-effective streaming of video games from the cloud with low latency. *IEEE Transactions on Circuits and Systems for Video Technology* 24, 8 (2014).

[34] Yu Wu, Chuan Wu, Bo Li, Xuanjia Qiu, and Francis CM Lau. 2011. Cloudmedia: When cloud on demand meets video on demand. In *31st International Conference on Distributed Computing Systems*. IEEE, 268–277.

[35] Fatos Xhafa. 2012. Data replication and synchronization in P2P collaborative systems. In *IEEE 26th International Conference on Advanced Information Networking and Applications*. IEEE, 7–7.

[36] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. 2010. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1–9.

[37] Yuhong Zhao, Hong Jiang, Ke Zhou, Zhijie Huang, and Ping Huang. 2014. Meeting service level agreement cost-effectively for video-on-demand applications in the cloud. In *IEEE Conference on Computer Communications 2014*. IEEE, 298–306.

[38] Hanying Zheng and Xueyan Tang. 2013. On server provisioning for distributed interactive applications. In *IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 500–509.