This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

Multi-user Computation Partitioning for Latency Sensitive Mobile Cloud Applications

Lei Yang, Jiannong Cao, *Senior Member, IEEE,*
Hui Cheng, *Member, IEEE,* and Yusheng Ji, *Member, IEEE*

Abstract—Elastic partitioning of computations between mobile devices and cloud is an important and challenging research topic for mobile cloud computing. Existing works focus on the single-user computation partitioning, which aims to optimize the application completion time for one particular single user. These works assume that the cloud always has enough resources to execute the computations immediately when they are offloaded to the cloud. However, this assumption does not hold for large scale mobile cloud applications. In these applications, due to the competition for cloud resources among a large number of users, the offloaded computations may be executed with certain scheduling delay on the cloud. Single user partitioning that does not take into account the scheduling delay on the cloud may yield significant performance degradation. In this paper, we study, for the first time, Multi-user Computation Partitioning Problem (MCPP), which considers the partitioning of multiple users' computations together with the scheduling of offloaded computations on the cloud resources. Instead of pursuing the minimum application completion time for every single user, we aim to achieve minimum average completion time for all the users, based on the number of provisioned resources on the cloud. We show that MCPP is different from and more difficult than the classical job scheduling problems. We design an offline heuristic algorithm, namely *SearchAdjust*, to solve MCPP. We demonstrate through benchmarks that *SearchAdjust* outperforms both the single user partitioning approaches and classical job scheduling approaches by 10% on average in terms of application delay. Based on *SearchAdjust*, we also design an online algorithm for MCPP that can be easily deployed in practical systems. We validate the effectiveness of our online algorithm using real world load traces.

Index Terms—mobile cloud computing; offloading; computation partitioning; job scheduling

1 Introduction

The proliferation of sensors on smart phones enables a new type of mobile applications, such as object/gesture recognition, mobile biometric, health-care monitoring or diagnosis, mobile augmented reality and so on. These applications often need continuous sampling and processing of high rate sensors like accelerometers, GPS, microphones and cameras. The compute-intensive classification algorithms of these applications usually lead to unsatisfactory performance on the compute-capability limited mobile devices. To address the problem, computation offloading has been proposed by researchers in recent years. The basic idea of computation offloading is to shift the execution of some tasks from the mobile device to cloud infrastructures. The powerful processors on cloud can run the compute-intensive tasks faster.

By using offloading technique, a fundamental problem is to partition the computations involved in the application between the mobile device and cloud, which is named as *computation partitioning problem*. Given that the application is composed of a set of dependent tasks, the computation partitioning problem is to decide whether each task is executed on the mobile device or on the cloud, such that the cost is minimized. The recent work [3]-[11] on the computation partitioning problem differ in their cost models. They consider either one of these factors such as energy consumption of the mobile device, application delay, and data transmission amount on the network, or the combination of these factors.

However, these work mainly focus on the partitioning problem under a user independent model, in which the computations are partitioned for one single user without regard to the partitioning results of other users. It is assumed that the cloud always has enough resources to accommodate without delaying the offloaded tasks, no matter how many other users offload the computations on the cloud. However, from the standpoint of the application provider, the assumption is not practical due to the following two reasons. First, the application providers need to balance the number of resources leased from cloud IaaS providers and the application performance, in order to lower their operational cost. Second, due to the unpredictable number of mobile users in large scale cloud applications, the application provider can not guarantee all the times to have enough resources to host the mobile users' offloading requests. Therefore, it is necessary to place the computation partitioning

- *Lei Yang and Jiannong Cao are with Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail: csleiyang@comp.polyu.edu.hk, csjcao@comp.polyu.edu.hk*
- *Hui Cheng is with School of Computing and Mathematical Sciences, Liverpool John Moores University, UK. E-mail: h.cheng@ljmu.ac.uk*
- *Yusheng Ji is with Information Systems Architecture Research Division, National Institute of Informatics, Japan. E-mail: kei@nii.ac.jp*

problem on constrained number of cloud resources. We name this problem as **Multi-user Computation Partitioning Problem (MCPP)**.

MCPP is much more challenging than the existing computation partitioning problems. In MCPP, the users' partitioning results are dependent with each other because of their competition for the cloud resources. For example, one user's decision on whether to offload the task not only depends on its saved computational cost and communication overhead, but also depends on how many other users offload the tasks onto cloud. The number of users who offload the tasks onto cloud represents the **load** on the cloud. If the load is high, the time spent in waiting for available cloud resources may sacrifice the benefit of offloading. An optimal solution for MCPP requires a unified schedule of all the users' computations onto their mobile devices and cloud resources.

In this paper, we study the MCPP for latency sensitive mobile cloud application. The problem is to schedule the offloaded computations on a constrained number of cloud resources as well as to partition the computations between mobile side and cloud side for all the users, such that the average application delay is minimized. The selected performance metric is average application delay/latency since it is the most critical one for latency sensitive mobile cloud applications. Moreover, we study how the application performance changes with the provisioned cloud resources and the load on the system, and thus construct a Performance-Resource-Load (PRL) model. The PRL model provides an optimal tradeoff between the application performance and the cost of cloud resources. We believe the model can help the application provider to achieve a cost-efficient utilization of the cloud resources, and hence save their operational cost. The main contributions of this work are as follows:

- To the best of our knowledge, this work is the first one to study the Multi-user Computation Partitioning Problem (MCPP), from the standpoint of application providers. The problem jointly considers the partitioning of computations for each user and the scheduling of offloaded computations on the cloud resources. It could help the application provider to achieve optimal application performance when faced with unpredictable number of users.
- We show that our MCPP is different from and more difficult than the existing job scheduling problems, such as Task Scheduling Problem in Heterogeneous Computing (TSPHC) and Hybrid Flow Shop (HFS) scheduling problems.
- We systematically solve the offline MCPP by proposing a set of competitive algorithms. Through the benchmarks, we show that our proposed algorithm, *SearchAdjust*, has better performance than the existing list scheduling algorithms by 10 percents in term of application delay.
- We design an online algorithm for MCPP that can be deployed in practical systems, and demonstrate its effectiveness using real world load traces.

2 Related Work

In this section, we briefly present the related work on computation partitioning problem in mobile cloud computing. Other related work on job scheduling problems are discussed in Section 3.4.

The application of cloud services in the mobile ecosystem enables a newly emerging mobile computing paradigm, namely *Mobile Cloud Computing* (MCC). We have classified three MCC approaches in our previous work [3]: a) extending the access to cloud services to mobile devices; b) enabling mobile devices to work collaboratively as cloud resource providers [1][2]; c) augmenting the execution of mobile applications using cloud resources, e.g. by offloading selected computing tasks required by applications on mobile devices to the cloud. This will allow the mobile developer to create applications that far exceed traditional mobile device's processing capabilities.

Most of the research work in mobile cloud use the third MCC approach [3]-[11][19]. They focus on the computation partitioning problem. In the work of [10][11], the computations are partitioned with a local view. The offloading decision for each task/module is made without the global view of the other tasks/modules involved in the application. For each task/module, if the reduced processing cost on the cloud is larger than the increased cost induced by data transmission across the network, then it is offloaded onto the cloud. [10] aims to save the energy consumption, while [11] aims to maximize the execution time and throughput.

In the work of [3]-[7][19], the computations are partitioned with a global view. The offloading decisions for each task/module are dependent with each other, and are jointly made with the profiling information about all the tasks/modules included in the application. [4] demonstrates that the approach of offloading with global view outperforms the approach with local view. The computation partitioning problem in this category is modeled as an optimization problem. They differ in the application model or the corresponding optimization objective. In [4][5], the application/program are modeled as a method graph/tree, where each vertex represents a method and is weighted with its computational and energy cost, and each edge represents method calling and is weighted with the size of data to be transferred remotely. The optimization problem is solved using Integer Linear Programming. [6][7] focus on the applications which are composed of a set of services, and aim to optimize one of factors such as latency, data transmission amount, and energy consumption,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

or optimize a customized combination of the factors above. [3] focuses on the partitioning of data stream application, and uses a dataflow graph to model the application. The genetic algorithm is used to maximize the throughput of the application.

We note that computation partitioning problem in all the work of [3]-[11][19] is solved from the standpoint of one single end user, with the assumption of unlimited resources at the cloud side. Few existing work has examined how to partition the computations when the number of users scales up, from the standpoint of application provider, with an aim to save the operational cost while maximizing the application performances experienced by end users. In this paper, we will study the multi-user computation partitioning problem with the consideration of the cloud resource constraint.

3 System model and Problem Formulation

3.1 Application model

We target for the *Latency Sensitive Mobile Cloud Applications*, which requires low latency for good user experiences. The applications often take sensory data as input, perform a sequence of operations onto the data, and then output the results. Mobile augmented reality is considered as one typical application. The application uses the camera and/or other sensors to perceive the user's environment/scene, and then augment the original scene with relevant information. The perception is done frequently which is driven by the user's input. The core part of augmented reality applications is the image based object recognition. Fig.1 shows the operations involved in the whole process of image based object recognition. Note that the SIFT algorithm is used to extract the features [12].

In our work, the applications are modeled as a sequence of processing modules (shown as vertex in Fig.1). The module represents a kind of operation onto the data. The directed edges represent the dependency between the modules. It means that a module can not start to run until its precedent module completes. Each module is allowed to run either locally on the mobile device or remotely on the cloud. Also the input data of the application is supposed to be from the sensors of the mobile device, and the output data should be delivered back to the mobile device. The performance metric is the execution time of the application. As the execution time represents the responsive delay/latency for the application, we simply use the term **delay** or **latency** in the paper. The delay is the summation of the computational time of all the modules and the data transmission time between the modules.

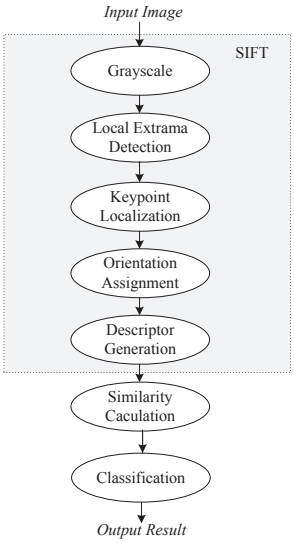


Fig. 1. The functional modules of image based object recognition

3.2 Single user computation partitioning

We first describe the *Single user Computation Partitioning Problem(SCPP)*, in which one single user runs the application and requests the cloud for computation offloading. Suppose the application consists of a sequence of n modules. Each module can be executed either at the mobile side or at the cloud side. The execution time of module j is c_j if it is offloaded onto cloud $(1 \leq j \leq n)$; otherwise, it is w_j, where $w_j > c_j$. If two adjacent modules j and $j+1$ run on different sides, the data transmission time is π_j; otherwise the data transmission time between j and $j+1$ becomes zero when they run on the same side. To model that the input/output data of the application should be from/to the mobile device, we add two virtual modules 0 and $n+1$ as the entry and exit modules.

Definition 1 *Single user Computation Partitioning Problem(SCPP)*: Given the computation cost c_j and w_j $(1 \leq j \leq n)$, and communication cost π_j $(0 \leq j \leq n)$, the SCPP is to determine which modules should be offloaded onto cloud such that the application delay is minimized. It is formulated by

$$\min_{x_j} d = \sum_{i=1}^{n} [(1-x_j)w_j + x_j c_j] + \sum_{j=0}^{n} |x_j - x_{j+1}| \pi_j, \tag{1}$$

where x_j is a binary decision variable. $x_j = 1$ if the module j is offloaded onto cloud, otherwise $x_j = 0$; and $x_0 = x_{n+1} = 0$.

3.3 Multiple users computation partitioning

Next, we illustrate the system model of multiple users computation partitioning. The system consists of two parts: cloud and mobile client. At mobile client, we have a set of users that send requests to cloud for partitioned execution of the application. The monitor

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

TABLE 1
Mathematical notations in this paper

j	index of module of the application;
n	total number of modules that the application contains;
c_j	execution time of module j at the cloud side;
w_j	execution time of module j at the mobile device;
π_j	data transmission time between module j and module $j+1$;
t_j	the completion time of module j;
x_j	decision variables in SCPP that indicate whether module j is offloaded onto cloud;
$t_j^{(c)}$	the completion time of module j if it is scheduled at the mobile side;
λ	number of user's requests;
r	number of cloud servers;
i	index of the user;
k	index of the machine that could be the mobile device or the cloud server;
T	the length of time interval;
δ_i	release time of user i's request;
(i,j)	the j-th module for the user i;
$w_{i,j}$	execution time of module j on the mobile device of user i;
$\pi_{i,j}$	data transmission time from module j to $j+1$ for user i;
$\tau_{i,j}$	the start time of module (i,j);
$t_{i,j}$	the completion time of module (i,j);
$x_{i,j,k}$	binary variable that indicates if module (i,j) is executed on server k;
$y_{i,j}$	binary variable that indicates if module j and $j+1$ of user i are executed on different sides k;
$z_{i,i',j,j'}$	binary variable that indicates if the execution of module (i,j) precedes module (i',j');
Δt	the length of time slot;
η	the index of time slot;
λ_η	the number of user's requests at time slot η;
$\overline{\lambda}$	the expectation value of λ_η over all time slots;
var	the variance value of λ_η over all time slots;
r_η	the number of cloud servers allocated to server user's requests at time slot η;

agent of the client middle-ware collects information on the device and wireless channel conditions. These information is sent with the requests to the cloud. The PaaS middleware at the cloud provides programming and run time support for partitioning and execution of the applications. Upon receiving users' requests, the partitioner of the PaaS middleware is to make the partitioning decisions for each user, i.e., to decide which modules of the application are executed on the mobile device and which modules are offloaded to the cloud, and also to schedule the offloaded modules onto the cloud servers/VMs. In our model, we assume that the users are requesting for partitioned execution of the same application. However, we can extend the model by considering that the users request for various applications, and easily apply the methods in Section 5,6,7 into the extended model. In our paper, we focus on the study of the partitioning problem and the design of corresponding algorithms that are implemented in the partitioner.

We consider a fixed time period $(0,T)$, during which a total number λ of requests are sent to the

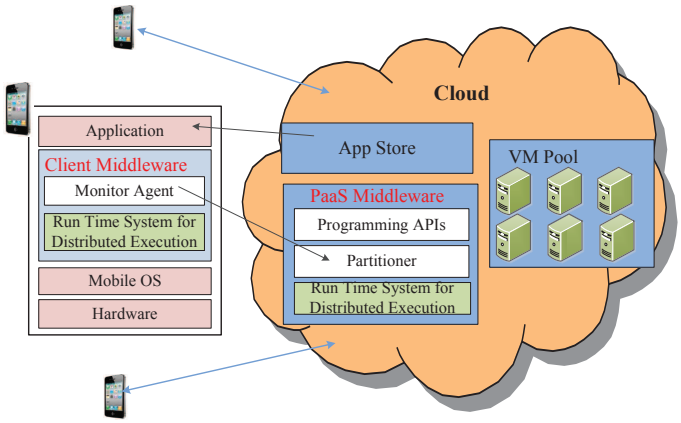


Fig. 2. System model of multi-user computation partitioning

cloud. Let i denote a particular request and δ_i denote the release time for request i, where $1 \leq i \leq \lambda$ and $0 \leq \delta_i \leq T$. For convenience of description, when we mention user i or i-th user in this paper, we refer to the user who emits request i.

We model cloud resources as a set of servers/VMs. The number of cloud resources is denoted as r. As mentioned in the SCPP, c_j represents remote computation cost (time) of the j-th module of the application. Usually the mobile users have different processing capabilities and networking bandwidth. Thus, we use a $\lambda \times n$ matrix W to represent local computation cost in which each $w_{i,j}$ gives the execution time to complete the module j on the i-th user's mobile device. Π is a $\lambda \times (n+1)$ communication cost matrix in which each $\pi_{i,j}$ $(0 \leq j \leq n)$ represents the data transmission time from the module j to $j+1$ for the user i.

We first study the offline multi-user partitioning problem, in which we assume perfect knowledge on the requests released from time 0 to T. We develop an offline algorithm as well as a set of competitive benchmark algorithms in Section 4 and 5. Based on the offline solutions, we design an online solution in Section 6.

Definition 2 *Multi-user Computation Partitioning Problem(MCPP)*: Given λ, r, δ_i, c_j, $W_{\lambda \times n}$ and $\Pi_{\lambda \times (n+1)}$, the problem is to determine for all the users at which machine (including the mobile device and the cloud servers) and at what time each module is executed, such that the average application delay of the users is minimized.

We formulate MCPP as a Mixed Integer Linear Programming(MILP) problem. In this formulation, we define one continuous variable $t_{i,j}$ and three 0-1 discrete variables $x_{i,j,k}$, $y_{i,j}$, $z_{i,i',j,j'}$. Please refer to Table 1 for their meaning. $x_{i,j,k} = 1$ if the module (i,j) is executed on the machine k, and otherwise $x_{i,j,k} = 0$. Note that $k = 1, 2, ..., r$ represents the servers at the cloud side, and $k = 0$ represents the mobile device. $y_{i,j} = 1$ if the two dependent modules, (i,j) and $(i,j+1)$, are executed on different sides, and otherwise

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

$y_{i,j} = 0$ if the two modules are on the same side; $z_{i,i',j,j'} = 1$ if the execution of module (i,j) precedes the module (i',j'), and otherwise $z_{i,i',j,j'} = 0$. We also define a positive constant IN which is as great as infinity. The MILP formulation can be written by Equation (2).

$$
\begin{aligned}
&\text{Min} \frac{1}{\lambda}\sum_{i=1}^{\lambda}(t_{i,n+1} - t_{i,0});\\
&\text{Subject to:}\\
&\text{(a) } \sum_{k=0}^{r} x_{i,j,k} = 1, \forall i \in [1,\lambda], \forall j \in [0,n+1];\\
&\text{(b) } t_{i,j+1} \geq t_{i,j} + \pi_{i,j}\cdot y_{i,j} + w_{i,j+1}\cdot x_{i,j+1,0}\\
&\qquad +c_{j+1}\cdot(1 - x_{i,j+1,0}), \forall i \in [1,\lambda], \forall j \in [0,n];\\
&\text{(c) } t_{i',j'} \geq t_{i,j} + c_{j'} - IN \times (1 - z_{i,i',j,j'})\\
&\qquad\qquad -IN \times (2 - x_{i,j,k} - x_{i',j',k}),\\
&\qquad\qquad \forall i,j,(i,j) \neq (i',j'), \forall k \in [1,r];\\
&\text{(d) } y_{i,j} \geq x_{i,j,0} - x_{i,j+1,0}, \forall i \in [1,\lambda], \forall j \in [0,n];\\
&\text{(e) } y_{i,j} \geq x_{i,j+1,0} - x_{i,j,0}, \forall i \in [1,\lambda], \forall j \in [0,n];\\
&\text{(f) } y_{i,j} \leq x_{i,j,0} + x_{i,j+1,0}, \forall i \in [1,\lambda], \forall j \in [0,n];\\
&\text{(g) } y_{i,j} \leq 2 - x_{i,j,0} - x_{i,j+1,0}, \forall i \in [1,\lambda], \forall j \in [0,n];\\
&\text{(h) } z_{i,i',j,j'} > (t_{i',j'} - t_{i,j})/IN, \forall i,j,(i,j) \neq (i',j');\\
&\text{(i) } z_{i,i',j,j'} \leq 1 + (t_{i',j'} - t_{i,j})/IN, \forall i,j,(i,j) \neq (i',j');\\
&\text{(j) } x_{i,j,k}, y_{i,j}, z_{i,j,i',j'} \in \{0,1\}, t_{i,j} \geq 0, \forall i,j,k,i',j'\\
&\text{(k) } x_{i,0,0} = 1, x_{i,n+1,0} = 1, t_{i,0} = \delta_i, \forall i \in [1,\lambda].
\end{aligned}
\tag{2}
$$

Constraint (b) guarantees the temporal order for the execution of two dependent modules. Constraint (c) indicates that each cloud server can and only can process one module at one time. In another word, if two modules are scheduled to the same machine, one module will not be started until the other one is finished. Since $y_{i,j}$ and $z_{i,i',j,j'}$ are two auxiliary variables, constraints (d)-(g) show that variables $y_{i,j}$ is determined by $x_{i,j,k}$, and constraints (h)-(i) indicate the value of $z_{i,i',j,j'}$ depends on the value of $t_{i,j}$ and $t_{i',j'}$. According to our system model, the input data of the application is from mobile device, so we have $x_{i,0,0} = 1$ for all i in constraint (k). Note that $t_{i,0}$ represents the *release time* of request i. Thus, we have $t_{i,0} = \delta_i$ in constraint (k).

3.4 Uniqueness of MCPP

We compare the MCPP with classical job scheduling problems and discuss their differences.

Comparison with TSPHC. The first classical scheduling problem similar to MCPP is Tasks Scheduling Problem for Heterogeneous Computing(TSPHC) [14]. In this problem, an application is represented by a directed acyclic graph(DAG) in which nodes represent application tasks and edges represent intertask data dependencies. Given a heterogeneous machine environment, where the machines have different processing speed, and the data transfer rate between machines are different, the objective of the problem is to map tasks onto the machines and order their executions so that task-precedence requirements are satisfied and a minimum completion time is obtained. The TSPHC is NP-complete in general case, and various efficient heuristics were proposed in the literatures [14][21].

Intuitively, we may model our problem as similar to MCPP as possible. In our problem, we have $\lambda \times n$ tasks, where the precedence dependence exists among the tasks from the same users. The machines can be abstracted as a set of r cloud servers/VMs and one mobile device. Note that the sole mobile device in our model, unlike the machines in TSPHC, is able to execute more than one task simultaneously. The data transfer rate is infinite between the cloud VMs , while being constrained between any pair of the mobile device and cloud VM. The problem is to map the tasks onto the $(r+1)$ machines such that the precedence constraints are satisfied, and the weighted summation of all the tasks' completion time is minimized. The tasks that appear in the last position of the application flow are assigned the weight of one, and others are assigned the weight of zero.

The key difference between MCPP and TSPHC is the optimization objective. In TSPHC the optimization objective is the makespan which is the maximum completion time of all the tasks, while in MCPP the objective is the total weighted completion time. Although various efficient heuristics were proposed for TSPHC to optimize the makespan, there were few solutions on optimizing the total weighted completion time. We can only find some early efforts to minimize the total weighted completion time on single machine or on parallel machine without considering the communications. Even these simplified versions for MCPP have been proved to be NP-hard [15].

Comparison with HFS. The second classical scheduling problem is Hybrid Flow Shop (HFS) scheduling [16]. In this problem, the job is divided into a series of stages. There are a number of identical machines in parallel at each stage. Each job has to be processed first at Stage 1, then Stage 2, and so on. At each stage, the job requires processing on only one machine and any machine can do. Assuming all the jobs are released at the beginning, the problem is to find a schedule to minimize the makespan. We note that the application and its functional modules in our problem are analogous to a job and stages in HFS. The mobile devices and cloud VMs may be modeled as the machines in HFS. However, our MCPP is far different from HFS in terms of the following aspects. 1) In MCPP, there exists communication overhead between stages, which makes the problem more complex than HFS; 2) in MCPP, since both cloud VM and mobile device are able to execute any module of the application, the set of machines are not partitioned into subsets

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

according to the stages; 3) the objective in MCPP is the total completion time rather than the makespan.

4 SEARCHADJUST

The most widely used method for solving MILP problems is branch and bound [17]. It transfers the MILP problem into standard Linear Programming (LP) problem by relaxing the integral variables. Based on the optimal solution obtained using LP, the MILP problem is then divided into subproblems by restricting the range of the integral variables. The subproblem is solved using LP, and then divided into subsubproblems. This process is done recursively until a feasible and satisfactory solution is found.

Unfortunately the LP-based solution is not practical for the MCPP, because it contains an exploding number of variables and constraints when the problem scales up. From equation (2), we can see that the number of variables $z_{i,i',j,j'}$ achieves the magnitude of $\lambda^2 n^2$, and the number of constraints (c) is $\lambda^2 n^2 r$. Although we can express the model by deleting all the auxiliary variables $z_{i,i',j,j'}$ and $y_{i,j}$, the number of variables $x_{i,j,k}$ remains a large magnitude of $\lambda n r$. Thus, in this section, we design a greedy heuristic algorithm, named as *SearchAdjust*, to solve the MCPP.

4.1 Overview

The idea of SearchAdjust is that we first relax the resources constraints in the MCPP. For each user, we can have an optimal partitioning by using the solution of SCPP. Under these optimal partitions, we *search* the time intervals during which the resources constraints are violated. We then *adjust* the schedule in a greedy way that can release as long resources occupation period as possible at these time intervals, and meanwhile increase the average application delay as little as possible. The *searching* and *adjusting* are done alternatively until the resource constraints are satisfied for all the time. The algorithm is designed due to the observation that the SCPP initiated solution is optimal but not feasible in the solution space of MCPP. Hence, we can adjust the initial solution iteratively to make it feasible, while not sacrificing the objective function (average application delay) too much.

Before describing the algorithm, we first introduce three important data structures as below.

(1) *Execution Schedule* S: For each user i, we can create a $n \times 3$ table to store its *execution schedule* in which each row is a three-tuple $(x_{i,j,0}, \tau_{i,j}, t_{i,j})$, where as described in Section 3.3 $x_{i,j,0}$ indicates at which side the module (i,j) is scheduled, and $\tau_{i,j}$, $t_{i,j}$ are respectively the start time and completion time of the module (i,j).

(2) *Cloud Resource Occupation List* L_{cro}: The list records the number of occupied servers at each time interval. Each element e of the list is denoted as $(start, end, num)$, where $start/end$ represents the start/end point of the time interval, and num is the number of occupied server at the interval. The time intervals in the list have no overlapping with each other, and are able to constitute a continuous time interval. The elements are stored in the list according to the ascending order of the time intervals. Thus, we have $e_k.end = e_{k+1}.start$ and $e_k.start < e_{k+1}.start$, $\forall k \geq 1$. Note that the length of each interval is not necessary to be the same.

(3) *Module Adjustment List* L_{adj}: The list records the modules which could release the cloud resource occupation period by waiting to execute later on or changing to run at mobile side, their rewards, and corresponding released cloud resource occupation period for the adjustment. We denote each item as $(i, j, reward, D_{rel})$, where i, j indicates the module, $reward$ represents the reward of the adjustment on this module and D_{rel} is the released cloud resource occupation period. The modules are stored in the list by a descending order of $reward$.

Algorithm 1 gives the pseudo code of the greedy heuristic. First, we get the optimal partitioning and corresponding execution schedule for each user without considering the cloud resource constraint (line 1). Second, we compute the cloud resource occupation list L_{cro} (line 2). It records the number of the occupied/in-use servers at each time interval. Then, we find the earliest interval that the number of occupied servers exceeds the up-bound r (line 3). We name the start of the interval as **critical point** t_{cri} on the time axis, as before it the resource constraint is satisfied, and after it the constraint is violated. Next, for each user we look for the module that was scheduled at the cloud side, and the execution time of which spans the critical point. In order to release the cloud resource immediately after the critical point, we adjust the schedule of this module by moving it back to mobile side or delaying its execution for some time. The adjustments are scored based on a reward function (which represents the greedy strategy in our algorithm) (line 6). The modules with positive score/reward are added into the module adjustment list L_{adj} (line 7-9). After the searching for all the users, α modules with largest rewards are selected from the list L_{adj} to adjust (line 12). In each iteration, the critical point t_{cri} would be moved forward along the time axis. The algorithm stops until the resource constraint is satisfied at all the times (line 3).

4.2 Details of SearchAdjust

In the following, we present details of *SearchAdjust*: 1) how to obtain the initial optimal but infeasible solution; 2) how to compute the cloud resource occupation list L_{cro} and search the critical point; 3) the reward function of *SearchAdjust*; 4) how to determine the number of modules α to adjust in each iteration.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

Algorithm 1: The Greedy Heuristic for MCPP

Input : A set of λ users, and a set of r cloud servers
Output: The execution schedule $(x_{i,j,0}, \tau_{i,j}, t_{i,j})$

```
Compute the initial execution schedule using SCPP solution;
Compute the cloud resource occupation list L_cro ;
while search the critical point from L_cro do
    for each user do
        if find the module that is scheduled onto cloud, and its execution time cover the critical point then
            Compute the reward of adjusting the module;
            if Reward > 0 then
                Insert this module into list L_adj by a descending order of its reward;
    Select the first α modules from list L_adj to adjust;
    Update the execution schedule of the selected modules;
    Re-compute the cloud resource occupation list L_cro;
return the execution schedule for all the users;
```

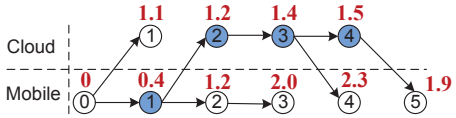
4.2.1 Initial Solution

Consider the SCPP shown in Definition 1, suppose that the completion time of module j is denoted as t_j. As every module can be completed either at mobile side or at the cloud side, we use notation $t_j^{(c)}$ to represent the completion time of module j if it is scheduled at the cloud side. Correspondingly, notation $t_j^{(m)}$ is the completion time of module j if scheduled at mobile side. Then, we have a recursive formulation of t_j:

$$t_j^{(c)} = min\{t_{j-1}^{(c)} + c_j,\ t_{j-1}^{(m)} + c_j + \pi_{j-1,j}\}, \quad (3)$$
$$t_j^{(m)} = min\{t_{j-1}^{(m)} + w_j,\ t_{j-1}^{(c)} + w_j + \pi_{j-1,j}\}, \quad (4)$$

where $j = 1, 2, ..., n, n+1$. The module 0 and module $n+1$ are respectively the entry and exit module we have added virtually into the application graph. The computation time of these two modules are zero.

In order to determine the optimal partitioning, we construct a graph which contains $2 \times (n+1)$ nodes. Each node is denoted as $v_j^{(p)}$, and labeled with its completion time $t_j^{(p)}$, where $0 \leq j \leq n+1$ and $p \in \{c, m\}$. Since the input data of the application is from the mobile device, we let $t_0^{(m)} = 0$ and $t_0^{(c)} = \infty$. Starting from the nodes $v_0^{(m)}$ and $v_0^{(c)}$, and we can recursively compute the labels of all the nodes by equation (3)(4). For each node, for example, $v_j^{(m)}$, there are two possible edges from its precedent nodes to it, $v_{j-1}^{(c)}$ and $v_{j-1}^{(m)}$. The edge that leads to less value of $t_j^{(m)}$ according to equation (4) is added into the graph. The partitioning result is actually a path from node $v_0^{(m)}$ to node $v_{n+1}^{(m)}$. Fig.3 shows an example of the method. There are four modules in the application. The colored nodes indicate the places that the modules are scheduled to. For the MCPP, with the optimal partitions $x_{i,j,0}$ for each user i, we can easily obtain the initial *execution schedule* $S = \{(x_{i,j,0}, \tau_{i,j}, t_{i,j})\}$.



$\{c_1, c_2, c_3, c_4\} = \{0.1, 0.2, 0.2, 0.1\}$;
$\{w_1, w_2, w_3, w_4\} = \{0.4, 0.8, 0.8, 0.4\}$;
$\{\pi_{0,1}, \pi_{1,2}, \pi_{2,3}, \pi_{3,4}, \pi_{4,5}\} = \{1.0, 0.6, 0.5, 0.5, 0.4\}$.

Fig. 3. An example of SCPP solution

4.2.2 Computation of Cloud Resources Occupation List and Critical Point

The cloud resource occupation list L_{cro} records the number of occupied server at each time interval. In each iteration of Algorithm 1, L_{cro} needs to be re-computed. We design an algorithm to calculate L_{cro}. The input of the algorithm is the *execution schedule* of each user $\{(x_{i,j,0}, \tau_{i,j}, t_{i,j})\}$. In the algorithm, L_{cro} is first initialized by the time interval $(0, \infty)$, with the number of occupied servers at this interval being zero. For each module (i, j) that is allocated to the cloud, we first respectively search from L_{cro} the interval in which the start and completion time of the module's execution period are located. The interval covering the start or the completion point is split into new sub-intervals, which are then inserted into L_{cro}. For the interval which is entirely covered by module (i, j)'s execution period, we increase the number of occupied servers by one. The algorithm stops until all the modules on cloud are finished. After finishing one module, the length of L_{cro} increases at most by 2. The length of L_{cro} returned by the algorithm would be at most $2\lambda n$.

4.2.3 Reward Functions/Greedy Strategies

The reward function is to evaluate the reward of adjusting the schedule of one module. It is defined by the *released cloud resource occupation period*, denoted as D_{rel}, minus the *extra delay* caused by this adjustment, denoted as D_{delay},

$$\textbf{Reward} = D_{rel} - D_{delay}. \quad (5)$$

The reward function is defined due to the motivation that we always prefer to select the module to adjust which can release as a long cloud resources occupation period as possible, and meanwhile causing as short extra delay as possible. Next we describe the definition of D_{rel} and D_{delay}.

Released Cloud Resource Occupation Period. Note that only the modules satisfying the following two conditions could be adjusted: (a) the module is executed at the cloud side, $x_{i,j,0} = 0$; and (b) its execution duration covers the critical point, $\tau_{i,j} < t_{cri} < t_{i,j}$. For one user i, assuming the module j_0 is the candidate module which cloud be moved to mobile side. To distinguish with the original *execution schedule* of user i, we use $\tau'_{i,j}$ and $t'_{i,j}$ to respectively represent the start time and completion time after the movement of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

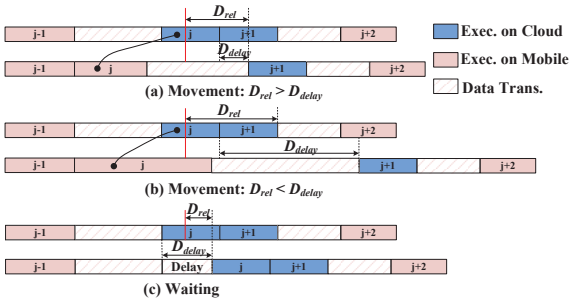


Fig. 4. Reward function

module j_0. The *released cloud resource occupation period* due to the adjustment of module (i, j_0) is defined by

$$D_{rel}(i, j_0) = min\{\tau'_{i,j_c}, t_{i,j_m-1}\} - t_{cri} \quad (6)$$

where $j_c, j_m \in [j_0 + 1, n + 1]$. j_c represents the first successor of module j_0 that is scheduled to the cloud; if no successor of module j is at the cloud side, then $\tau'_{i,j_c} = \infty$. j_m is the first successor of module j_0 that is scheduled onto the mobile side; if no successor of module j_0 is at mobile side, then $j_m = n + 1$.

Extra Delay. The extra delay caused by the adjustment of module (i, j_0) is defined by

$$D_{delay}(i, j_0) = \tau'_{i,j_0+1} - \tau_{i,j_0+1} \quad (7)$$

The reward of adjustment of one module could be positive or negative. Fig.4(a) indicates that $D_{rel} > D_{delay}$, hence the reward of the adjustment is positive; while in Fig.4(b) the reward of the adjustment is negative. In each iteration of algorithm, we only select the modules with positive and as large as possible reward to adjust.

Other Reward Functions. Now we pose another question: do we have other reward functions? Actually there exist two typical functions: (a) **Reward** $= -D_{delay}$, and (b) **Reward** $= D_{rel}$. The former function means that the modules with as small extra delay as possible are selected despite of its released cloud resource occupation duration. The latter function prefers to select the module that could release longer cloud resource occupation duration. We evaluate the two functions in Section 7. We find that function (a) obtains good average application delay, but requires a long time to converge, while function (b) leads to bad average application delay. The reward function in Equation (5) is able to achieve good average delay and fast convergence speed.

Adjusting Options. Remember that we actually have two adjusting options to release the resources which have been occupied at the critical point, i.e., **waiting** and **movement**. Waiting means to purely delay the execution of the modules at the cloud side, while movement means to change the execution place of the module. Fig.4 illustrates the reward functions under the two adjusting options. The three colors of the bar represent three different execution procedures of the application graph, i.e., local execution, remote execution and data transmission. The length of the bar represents how much time the procedure takes. The completion time of the application graph equals to the total length of all the bars in various colors. Fig.4(a)(b) shows the option of moving module j from the cloud side to the mobile side. It can release certain cloud occupation time D_{rel} and cause extra application delay D_{delay}. Fig.4(a) illustrates the case of $D_{rel} > D_{delay}$. Fig.4(b) illustrates the case of $D_{rel} < D_{delay}$. Fig.4(c) shows the option of waiting. We can see that waiting adjustment always has a non-positive reward. Hence, to simplify the Algorithm 1, we use the moving adjustment, and do not consider the waiting adjustment.

4.2.4 Number of modules to adjust α

Now we answer the question: how many modules are adjusted in each iteration? Note that in Algorithm 1, only the modules with positive rewards are put into the L_{adj}. First, we get average D_{rel} of all the modules in L_{adj}, which is denoted as $D_{rel}^{(avg)}$. Then, from the cloud resource occupation list L_{cro}, we compute the average number of occupied servers in the period from t_{cri} to $t_{cri}+D_{rel}^{(avg)}$, which is denoted as $num^{(avg)}$. The number of modules to adjust α is given by Equation (8-10):

$$\alpha = \min\{\text{LengthOf}(L_{adj}), num^{(avg)} - r\}, \quad (8)$$

$$num^{(avg)} = \frac{\sum_{e \in L_{cro}^{(sub)}} (e.end - e.start) \times e.num}{\sum_{e \in Sub\{L_{cro}\}} (e.end - e.start)}, \quad (9)$$

where $L_{cro}^{(sub)}$ is the subset of L_{cro}, which includes all the time intervals located at $[t_{cri}, t_{cri} + D_{rel}^{(avg)}]$,

$$L_{cro}^{(sub)} = \{e \in L_{cro} | e.start, e.end \in [t_{cri}, t_{cri} + D_{rel}^{(avg)}]\}. \quad (10)$$

In our algorithm, the adjustment on the schedule usually leads to the decreasing of application performance. We avoid the case that excessive modules are moved back to mobile side, such that the cloud servers are not utilized completely. So α is constrained by an up-bound $num^{(avg)} - r$ as shown in Equation (8).

4.3 Theoretical Analysis

In Algorithm 1, the execution schedules indicate if each module is executed at mobile side or at the cloud side. However, for the modules that are allocated to the cloud, the results do not specify which cloud server hosts the offloaded module. We may question: could the completion time of each module $t_{i,j}$ be delayed when allocating the offloaded modules onto the cloud servers?

Theorem 1 (Feasibility) For the execution schedule $S = \{(x_{i,j,0}, \tau_{i,j}, t_{i,j})\}$ generated by Algorithm 1, we can always find a feasible schedule $S' =$

$\{(x'_{i,j,k}, \tau'_{i,j}, t'_{i,j})\}$ of MCPP by assigning the offloaded modules onto the cloud servers, such that each module (i,j) is completed no later than $t_{i,j}$, $t'_{i,j} \leq t_{i,j}$.

Proof: Consider a simple case where the offloaded modules are scheduled online to the cloud servers with the policy of 'first-come-first-serve'. $\tau_{i,j}$ can be taken as the time that the module (i,j) comes to the cloud. We can prove using mathematical induction, by the 'first-come-first-serve' policy every offloaded module (i,j) can start exactly at time $\tau_{i,j}$ and be completed by the time $t_{i,j}$ on the cloud servers. For the new schedule S', we have $\tau'_{i,j} = \tau_{i,j}$, and $t'_{i,j} = t_{i,j}$.

The offloaded modules are ordered according to their arriving time $\tau_{i,j}$. For the 1-st module which comes to the cloud earliest, obviously it is able to start at its arriving time τ_{i_1,j_1}. For the s-th module which comes to the cloud in s-earliest time, we can prove if 1-st, 2-nd, ... , $(s-1)$-th modules start to run at their arriving time, then the s-th module can also be executed at the cloud servers at its arriving time τ_{i_s,j_s}. The proof is as follows.

Up to the time when the s-th module comes, we can conclude: 1) at least one cloud server is idle at $\tau_{i,j}$; 2) if the cloud server is idle at time τ_{i_s,j_s}, it must be idle all the time $[\tau_{i_s,j_s}, +\infty)$. The first conclusion is due to the fact, that our algorithm guarantees the number of occupied cloud servers does not exceed the constraint r at all the times if each module is executed according to the schedule S. The second conclusion is proved in this way: if 2) does not hold, which means that up to the time τ_{i_s,j_s}, some module which arrives at the cloud earlier than the s-th module has already been allocated to the cloud server. It contradicts with the 'first-come-first-serve' policy. Because of the two conclusions, the s-th module can start on the cloud server at its arriving time. If multiple idle servers exist, we randomly allocate the module to one of them. □

Theorem 2 (Complexity) For a given application graph, the complexity of Algorithm 1 is $O(\lambda^2)$.

Proof: In Algorithms 1, the evaluation of the reward for adjusting each offloaded module is the most time costly operation. In each iteration, at most $\lambda \times n$ modules need to be evaluated. The question is how many iterations Algorithm 1 needs to stop. The worst case is that the resources constraints $r = 0$, and only one module is moved to the mobile side in each iteration. In this case, Algorithm 1 needs at most $\lambda \times n$ iterations, such that the offloaded modules are all moved to the mobile side. By neglecting the constant n for a given application graph, the worst time complexity of Algorithm 1 is on the order of $O(\lambda^2)$. □

5 Benchmark Offline Solutions

List scheduling is considered as an efficient method to solve existing job scheduling problems [16][20][22][23]. Although MCPP is different from existing job scheduling problems, we are still interested to know how list scheduling based algorithms perform when used to solve the MCPP. We have two ways to solve the MCPP using list scheduling method. One way is, as described in 3.4, we can fit the MCPP into the TSPHC model by abstracting both the mobile devices and cloud servers as the processors. The problem can be solved by Heterogeneous-Earliest-Finish-Time (**HEFT**) algorithm, which is demonstrated to be an accurate and efficient list scheduling algorithm for TSPHC [14]. The time complexity of HEFT is on the order of $O(\lambda \times r)$. We will evaluate the performance of HEFT algorithm when it is used to solve the MCPP in Section 7.

Algorithm 2: The MEDLS Heuristic

Input : The execution schedule $(x_{i,j,0}, \tau_{i,j}, t_{i,j})$
Output: The execution schedule $x_{i,j,k}, \tau_{i,j}, t_{i,j}$

```
for each user i do
    Insert the first offloaded module of the user into a scheduling list L_s;
Sort the modules in L_s by non-increasing order of their ready time τ_{i,j};
while there are unscheduled tasks in the list L_s do
    Select the first module (i_0, j_0) from the list for scheduling;
    for each machine k, including cloud servers and the user's mobile device do
        Compute the extra delay of module (i_0, j_0) on machine k.;
    Assign task (i_0, j_0) to the machine that minimizes the extra delay;
    Update the execution schedule for user i_0, (x_{i_0,j,k}, τ_{i_0,j}, t_{i_0,j});
    Remove the module (i_0, j_0) from L_s, and add the first successive offloaded module of (i_0, j_0) into L_s;
return the execution schedule x_{i,j,k}, τ_{i,j}, t_{i,j};
```

The other way is that we divide the MCPP into two phases. In the first phase, named as partitioning, we simply decide for each user which modules are executed at mobile side and which others are scheduled at the cloud side. The partitioning phase is done using the SCPP method, which is introduced in Section 4.2.1. In the second phase, the list scheduling method is applied to allocate the offloaded modules onto the cloud servers or to move the offloaded modules to the mobile side. Based on the generated execution schedule from the first phase, the task that is ready to start earliest is assigned with the highest priority. Each selected task will be scheduled to the machine (including the cloud servers and the mobile device) which leads to a minimum extra delay. We name the method as Minimum Extra Delay List Scheduling (**MEDLS**). Algorithm 2 gives the pseudo code of the MEDLS. The time complexity of the algorithm is also $O(\lambda \times r)$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

6 Online Solution

In contrast to the offline solutions, an online solution only knows the release time of past requests and current requests but have no knowledge about the future requests. The partitioning for one user's request can not be determined before the request is released. In this section, we present the design, and analyze the performance of our online solution.

In our online solution, we do not partition user's requests one by one. Instead, we divide the whole time interval $(0, T)$ into small time slots, and do the partitioning every time slot. Let η denotes the index of the time slot, and Δt denotes the length of the time slot. We do the partitioning at the end of each time slot for all the requests that are released during that time slot. For each partitioning, we first select a number of idle servers from all the r cloud servers, and then use our offline solution to do the partitioning with the selected cloud servers. The offline solution is performed repeatedly every Δt. Note that Δt is small enough relative to the completion time of the application.

Now the question here is that how many servers are allocated to the load at each time slot η, such that the overall delay for the requests during $(0, T)$ is as low as possible. If we allocate too many servers to current load, it is possible that there is no enough idle cloud servers to accommodate the load in future time slots. If we always try to reserve more servers to future load, the performance of current load would yield to significant degradation. The online algorithm tries to balance a tradeoff between provisioning enough servers for current load, and reserving enough servers for future load, through a control parameter Λ.

Let λ_η denote the number of requests that arrive at the system during time slot η. Let r_η denote the number of servers that are allocated to the requests at the end of time slot η. The overall delay of the λ_η requests are obtained by our PRL model $d_\eta = \mathcal{F}(\frac{r_\eta}{\lambda_\eta})$, which is numerically analyzed in Section 7.1.2. Suppose d_η is normalized by the length of the time slot Δt. Think in the way that in order to accommodate the load λ_η, r_η servers will be occupied for d_η time slots. Therefore, we define *workload size* W_η that arrives at the cloud servers at the end of time slot η by $W_\eta = r_\eta \times d_\eta = r_\eta \times \mathcal{F}(\frac{r_\eta}{\lambda_\eta})$. Note that the workload size W_η reduces by r_η at the end of time slot $\eta + 1$. Let Q_η denote the total backlogged workload size of the cloud servers at the end of time slot η, before any other loads arrive. Let D_η denote the number of servers that are busy/occupied at the end of time slot η, where $0 \leq D_\eta \leq r$. Then the dynamic of $Q_{\eta+1}$ can be described as

$$Q_{\eta+1} = Q_\eta + W_\eta - D_\eta. \tag{11}$$

Without loss of generality, we assume load λ_η is a stochastic process across time slot η. Let $\overline{\lambda}$ and var respectively denote the expectation value and variance of load sequence λ_η. We say that the system is stable if $\lim_{\eta\to\infty} E(Q_\eta) < \infty$, i.e., the amount of backlogged workload size is bounded. The arriving load λ_η is said to be supportable if there exists a resource allocation mechanism under which the system is stable.

Theorem 3 For any given $\overline{\lambda}$ and r, the expectation of application delay of the arriving load λ_η is upbounded by d_{opt}, where

$$\begin{aligned} d_{opt} &= \min_{r^* \geq 0} \mathcal{F}(\frac{r^*}{\overline{\lambda}}), \\ \text{s.t. } r^* &\times \mathcal{F}(\frac{r^*}{\overline{\lambda}}) < r. \end{aligned} \tag{12}$$

Proof: In order to support the arriving load, it should be satisfied that $\lim_{\eta\to\infty} E(Q_\eta) < \infty$. Thus, from Equation (11), we have $E(W_\eta) = E[r_\eta \times \mathcal{F}(\frac{r_\eta}{r_\eta})] \leq D_\eta$. D_η represents the *workload size* that the cloud can finish in time slot η. The maximum of D_η is equal to r, and can be achieved if and only if all the r cloud servers are busy to process the workload during time slot η. Thus, we have $E[r_\eta \times \mathcal{F}(\frac{r_\eta}{r_\eta})] \leq r$. Let $E(r_\eta) = r^*$, we then get Equation (12). □

The key aspect of our algorithm is that it manages a pool of idle servers. At each time slot, some servers are removed from the pool to accommodate the coming load, meanwhile some new servers become idle and are added into the pool. Suppose I_η is the number of idle servers at the end of time slot η before allocating the servers for load λ_η. The parameter I_η can be obtained by the recursive equation

$$I_{\eta+1} = I_\eta - r_\eta + R_{\eta+1}, \tag{13}$$

where r_η are the number of servers that are allocated to load λ_η, and $R_{\eta+1}$ are the number of servers which are newly added into the pool of idle servers at the end of time slot $\eta+1$. Note that we have $I_1 = r$ at the initial time slot.

The **online algorithm** first calculates the optimal overall delay d_{opt} and corresponding number of allocated servers r_{opt} at each time slot according to Equation (12), where $r_{opt} = argmin_{r^* \geq 0} \mathcal{F}(\frac{r^*}{\overline{\lambda}})$. At the end of each time slot η, the algorithm does the following:

- Compute the number of idle servers I_η by Equation (13).
- Compute the number of released servers in next time slot $R_{\eta+1}$ according to the delay of previous load $d_{\eta-1}, d_{\eta-2}, d_{\eta-3}$ ...
- Determine the number of servers to be allocated r_η. We are trying to guarantee that the delay of the load λ_η achieves d_{opt}. Thus, intuitively we would allocate $\frac{r_{opt}}{\overline{\lambda}} \times \lambda_\eta$ servers to the current load. If the number of idle servers is not enough to guarantee the delay d_{opt}, i.e., $I_\eta < \frac{r_{opt}}{\overline{\lambda}} \times \lambda_\eta$, we would allocate all the idle servers to current load, i.e., $r_\eta = I_\eta$; otherwise, we would consider the following rules:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

- **Rule 1** Allocating more servers to improve the performance at current time slot, i.e., $r_\eta \geq \frac{r_{opt}}{\overline{\lambda}} \times \lambda_\eta$.
- **Rule 2** Reserving enough servers for next time slot. The idle servers in next time slot $\eta+1$ would be enough to accommodate $(1+\Lambda)\overline{\lambda}$ load, i.e., $r_\eta < I_\eta + R_{\eta+1} - (1+\Lambda)\overline{\lambda}$.
- **Rule 3** Avoiding over-provisioning servers for current load, i.e., $r_\eta \leq C_{max}\lambda_\eta$, where C_{max} is a constant of our PRL model. If and only if $\frac{r_\eta}{\lambda_\eta} < C_{max}$, increasing the number of allocated servers r_η would lower the delay of current load (see Fig.5f); otherwise if $\frac{r_\eta}{\lambda_\eta} \geq C_{max}$, increasing r_η would not improve the performance of current load.

• If Rule 1 contradicts with Rule 2, i.e., $\frac{r_{opt}}{\overline{\lambda}} \times \lambda_\eta > I_\eta + R_{\eta+1} - (1+\Lambda)\overline{\lambda}$, then we give priority to Rule 1, in which case we have $r_\eta = \frac{r_{opt}}{\overline{\lambda}} \times \lambda_\eta$; otherwise we have $r_\eta = \min\{I_\eta + R_{\eta+1} - (1+\Lambda)\overline{\lambda}, C_{max}\lambda_\eta, I_\eta\}$. Note that Rule 1 and Rule 3 never contradicts with each other, because $\frac{r_{opt}}{\overline{\lambda}} \leq C_{max}$.

7 Evaluation

We will respectively evaluate the performance of the offline solutions and online solutions. Our online solution divides the time axis into a number of small time slots, in each of which the offline solution is applied to partition all the requests that arrive during that time slot. Since the time slot is small enough, the release time of all the requests in each time slot are the same. Therefore, for simplicity, in the evaluation of offline solution, the release time of all the requests are assigned as zero. Through the evaluation of various offline solutions, we aim to answer two questions: 1) which solution performs best; 2) how the application performance (delay) varies depending on the number of cloud resources and the load, i.e., Performance-Resource-Load (PRL) Model. The evaluation of online solution is then based on the PRL model and the real world wikipedia load traces [18].

7.1 Evaluation of Offline Solutions

We use the application of image based object recognition as shown in Fig.1 in our evaluation. It contains seven modules, therefore we have $n = 7$. We profile manually the execution time of each module on our laboratory server, and the data size that needs to be transferred between two connective modules. We assume that the processing time of each module on the mobile devices is F times greater than that on the server. Since the users' mobile devices have different processing capability, the users have various factor F. In our experiments, the local computation cost is generated by $W_{\lambda\times 7} = [F_1, F_2, ..., F_\lambda]^T \times \mathcal{C}$, where $\mathcal{C}$ is a 1×7 vector of the profiled execution time of each module on the server and F_i yields a uniform distribution in the interval $[1, 6]$. The communication cost is generated by $\Pi_{\lambda\times 8} = [\frac{1}{B_1}, \frac{1}{B_2}, ..., \frac{1}{B_\lambda}]^T \times \mathcal{D}$, where $\mathcal{D}$ is a 1×8 vector of the profiled data size, and B_i $(1 \leq i \leq \lambda)$ is the communication bandwidth which also yields a uniform distribution. We have generated more than 2000 test cases by changing the number of cloud servers r or the number of users (load) λ. Whenever λ is changed, $W_{\lambda\times 7}$ and $\Pi_{\lambda\times 8}$ need to be re-generated.

The comparison of various algorithms are based on the following two metrics:

• **Metric 1: Application Delay Ratio (ADR)**. The main performance measure of the algorithms is the average application delay that is experienced by the mobile users. Since a large set of tests are performed under different load λ and resources r, it is necessary to normalize the application delay to a lower bound, which is called the Application Delay Ratio (ADR). The ADR value of an algorithm is defined by

$$ADR = \frac{application\ delay}{d_{scpp}}. \quad (14)$$

The denominator is the application delay under the SCPP solution. In SCPP, the cloud resources are assumed to be unconstrained, and each user's execution schedule is generated independently by SCPP method. The ADR of the MCPP algorithms can not be less than one since the dominator is the lower bound. The MCPP algorithm that gives the lowest ADR is the best algorithm with respect to performance.

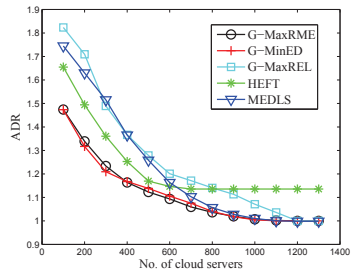
• **Metric 2: Running Time of the Algorithms**. The running time of an algorithm is its execution time for outputting the schedule. The metric gives the average cost of the algorithm. For the algorithms which have very close ADR values, the one with minimum running time is considered as the best one.

7.1.1 Performance of SearchAdjust

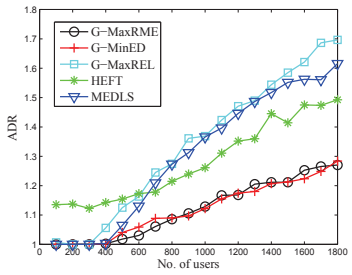
We compare the ADR performance of the greedy heuristic, SearchAdjust (Algorithm 1), under three different greedy strategies, with two list scheduling algorithms, HEFT [14] and MEDLS (Algorithm 2). A concise description about the algorithms is as follows.

- **G-MaxREL**. G-MaxREL is the greedy heuristic with the strategy to maximize the Released Cloud Resource Occupied Period. The reward function in the algorithm is $Reward = D_{rel}$.
- **G-MinED**. G-MinED is the greedy heuristic with the strategy to minimize the Extra Delay. The reward function is $Reward = -D_{delay}$.
- **G-MaxRME**. G-MaxRME is the greedy heuristic with the strategy to maximize Released Cloud Resource Occupied Period minus Extra delay. The reward function is $Reward = D_{rel} - D_{delay}$.
- **HEFT**. HEFT is a well-known list scheduling algorithm specifically for TSPHC problems. We can also use the algorithm to solve our MCP problem.

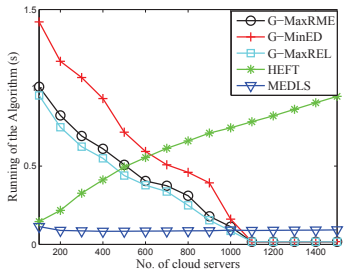
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers



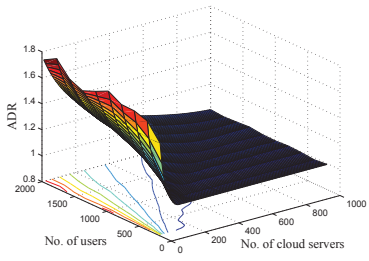
(a) Comparison of ADR based performance under various number of cloud servers ($\lambda = 2000$)



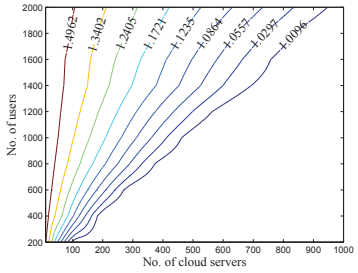
(b) Comparison of ADR based performance under various load ($r = 200$)



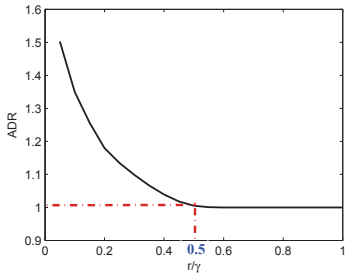
(c) Comparison of the algorithm running time



(d) The 3D Performance-Resource-Load (PRL) curve



(e) Contour lines of of PRL model



(f) The ADR performance varies depending on r/λ

Fig. 5. Evaluation results of offline solutions

- **MEDLS**. MEDLS is a list scheduling algorithm which is designed to solve the MCP problem [14]. In this algorithm, each module is scheduled to the machine which causes the Minimum Extra Delay.

In the first experiment, the number of users is fixed at $\lambda = 2000$. The ADR-based performance of the algorithms are compared with respect to various number of cloud servers (see Fig.5a). Among the three greedy heuristics, G-MaxRME and G-MinED achieve better performance than G-MaxREL. Compared with HEFT, the performance of our proposed greedy heuristics (G-MaxRME and G-MinED) is better for any number of cloud servers. Compared with MEDLS, the greedy heuristics (G-MaxRME and G-MinED) have better performance when the cloud resources is relatively tight ($r < 800$). When the cloud resources increase to $r > 800$, the greedy heuristics have the same performance with MEDLS, because in this case the SCPP solution used as the initial solution in both the greedy heuristics and MEDLS becomes feasible for MCPP. The average ADR value of the greedy heuristic (G-MaxRME or G-MinED) on all the numbers of cloud servers is better than the HEFT algorithm by 11 percent, and the MEDLS algorithm by 10 percent. It is also shown that, for all these five algorithms, the performance increases as the number of the cloud servers increases. It demonstrates the application providers can increase the overall application performance by leasing more cloud resources.

Next, we fix the number of cloud servers, $r = 200$, and compare the ADR performance of the five algorithms when the number of users λ varies (see Fig.5b). The two proposed greedy heuristics (G-MaxRME and G-MinED) outperform other algorithms in terms of the overall ADR performance under various number of users. For the greedy heuristics and MEDLS, there exists a threshold, $\lambda = 400$, below which the performance is not affected by the number of users. It is because in this case the cloud always has enough resources to accommodate the offloaded modules, such that each user can realize its SCPP based optimal partitions. However, when the number of users exceeds the threshold, it is shown that the performance degrades quickly as λ increases.

We compare the cost of the algorithms by using the metric of the algorithm running time (see Fig.5c). MEDLS is the least costly one among the five algorithms. For the two greedy heuristics (G-MaxRME and G-MinED) which have the best ADR-based performance, it is shown that G-MaxRME is more costly than G-MinED. This is because G-MaxRME includes the released cloud resource occupation time into the reward function, and hence needs fewer iterations than G-MinED. *We conclude that G-MaxRME is the best one among all the five algorithms in terms of both ADR based performance and running time.* Another interesting thing we can observe from Fig.5c is that, our proposed greedy heuristics have less running time as the cloud resources increase, while HEFT has longer running time as the cloud resources increase. This is because the greedy heuristics need more iterations to adjust the SCPP initial solution as the cloud resources constraint is lower. However, as the cloud resources increase, the running time of HEFT increases because

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers

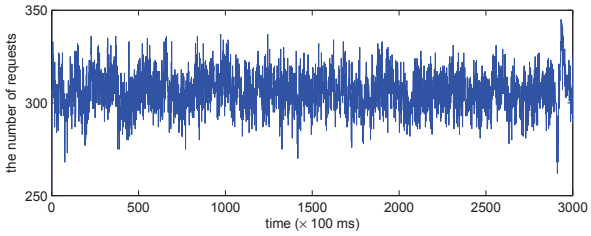


Fig. 6. One selected wikipedia load trace: it contains 3000 time slots. The expectation value and variance of this trace are $\overline{\lambda} = 305$, $var = 12$

more time is spent in machine selection with an insertion-based policy.

7.1.2 Performance-Resource-Load (PRL) Model

Finally, we evaluate how the ADR-based performance changes with the provisioning cloud resources r and load λ on the system by using our proposed greedy heuristic (G-MaxRME). We have measured the ADR value of G-MaxRME algorithm in about 2000 test cases, where the number of cloud servers r varies in $[10, 2000]$ and the number of users λ varies in $[200, 2000]$. Fig.5d and Fig.5e show a 3-dimension visual PRL curve and its corresponding contour lines. Note the contour line contains the (r, λ) points which has the same ADR value. The straight contour lines approximately from the origin $(0, 0)$ show that the ADR based performance depends on the ratio of r and λ. Based on this observation, we construct a mathematical PRL model $d = \mathcal{F}(\frac{r}{\lambda})$. Fig.5f shows the fitting curve of ADR and $\frac{r}{\lambda}$ values from our simulation results. The greater the r/λ value is, the better the performance is. When r/λ is more than about $1/2$, the performance is not affected by the increase of r/λ, because the number of cloud resources is equivalent to unlimited with respect to the number of requesting users.

7.2 Evaluation of Online Solution

To realistically evaluate the performance of our online solution, we use the wikipedia request traces [18] to do the simulations. The whole data set contains 10% of all the requests directed to Wikipedia server from September 19, 2007 to January 2, 2008. The total number of the requests are 20.6 billion. Each request in the data set includes a time stamp which is recorded in milliseconds. From the whole data set we select 10 traces, each of which has a length of 5 minutes. For each trace, we count the number of requests every one time slot (we set the length of time slot $\Delta t = 100ms$ in this simulation). Thus, each trace contains 3000 time slots. Note that to evaluate how the variance of the load trace influences our online algorithm, we select the 10 traces which have the same expectation value ($\overline{\lambda} = 305$) but different variances ($5 \leq var \leq 50$). Fig.6 shows one of the 10 selected traces. Table 2 shows the setting up of the parameters in our simulation.

TABLE 2
Parameters setting up for online algorithm

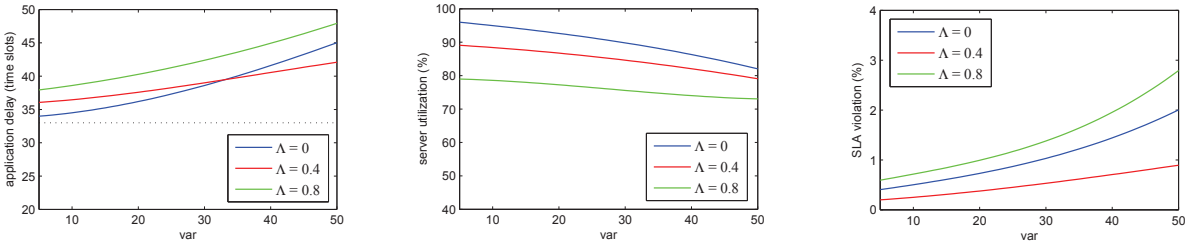
Parameters	Values
Length of time slot Δt	100 ms
Length of load trace N	3000 time slots
The expectation value of load trace $\overline{\lambda}$	305
The variance of load trace var	(5, 50)
The number of cloud servers r	3000
Control parameter of online algorithm Λ	0, 0.4

We evaluate our online algorithm in terms of the three metrics: (i) application delay, (ii) server utilization, and (iii) Service Level Agreement (SLA) violation. *Application delay* indicates the average delay of all the requests in the load trace. *Server utilization* is defined by $U = 1 - \frac{\sum_{\eta=1}^{N}(I_\eta - r_\eta)}{rN}$, where $N = \frac{T}{\Delta t}$ is the length of load trace in time slots and $I_\eta - r_\eta$ represents the number of servers that are spare during time slot η. *SLA violation* is defined as the percentages of requests that do not meet the delay requirements. For latency sensitive applications, the application provider requires that the delay of each request should be less than a constraint. If and only if the delay of all the requests meet the delay requirement, we say the SLA are satisfied, in which case the value of SLA violation is zero. Note the difference between metric (i) and metric (iii). The two metrics are not positively correlated. If the input load trace has low application delay, it does not necessarily have low SLA violation; and vice versa.

Fig.7 shows the evaluation results for multiple load trace variances and three values of Λ. It is shown that as the variance of input load trace var increases, the application delay and SLA violation generally increases and server utilization generally decreases. The reason is that as var increases, the load fluctuation over time has greater magnitude, resulting more time slots in which the cloud servers are either overloaded or mostly spare. In particular, Fig.7a demonstrates that the application delay of our online algorithm is very close to the optimum d_{opt} (shown as dash line) when the variance is small. For example, the ratio between the application delay ($var = 20$, $\Lambda = 0.4$) and d_{opt} is 1.08. The smaller var is, the closer the ratio value is to 1. We analyze the whole wikipedia request data set during October 2007, we found that more than 90% of the traces (3000 time slots length) has $var \leq 20$. This evaluation implies that our online algorithm can achieve a delay of $1.08d_{opt}$ at most time, i.e. 90% time of the whole month in October 2007 for wikipedia requests data set.

The trade-off between reserving more servers for future time slots ($\Lambda = 0.4$) and allocating more servers for current time slot ($\Lambda = 0$) is also interesting. Fig.7a shows that when the load trace has small variance, it is better to allocate more servers for current time; while the load trace has large variance, it is better to reserve more servers for future time. Although

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2366735, IEEE Transactions on Computers



(a) The overall application delay increases as load variance var increases

(b) The cloud server utilization varies depending on var

(c) SLA violation increases as var becomes larger

Fig. 7. The three key metrics for online algorithm on Wikipedia load traces

reserving more server for future time slots always leads to lower server utilization than allocating more servers for current time (shown in Fig.7b), it can achieve better performance in term of SLA violation (shown in Fig.7c). However, when the value of Λ is too large, i.e., $\Lambda = 0.8$, the algorithm does not achieve good performance in terms of all the three metrics.

8 Conclusion

In this paper, we have focused on the Multi-user Computation Partitioning Problem (MCPP). We have designed an offine algorithm, *SearchAdjust*, and a set of competitive benchmark algorithms to solve the problem, and conducted extensive simulations to compare their performance. *SearchAdjust* was demonstrated to outperform the list scheduling algorithms, HEFT and MEDLS, by 10 percent in term of the application delay. From the simulations, we also draw a Performance-Resource-Load (PRL) model to show how the performance (application delay) varies depending on the load and provisioned cloud resources. Based on the PRL model and offline algorithm, we further design an online algorithm that can be deployed in practical mobile cloud systems. We show our online algorithm can achieve satisfactory application delay by real trace driven simulations.

Acknowledgments

The research is supported by Hong Kong RGC under GRF (Grant No. c02461), and Microsoft (Grant No. H-ZD89).

References

[1] G. Canepa, and D. Lee. A Virtual Cloud Computing Provider for Mobile Devices. In *Proc. of ACM MCS*, 2010.
[2] E. E. Marinelli. Hyrax: Cloud Computing on Mobile Devices using MapReduce. In *Master Thesis, Carnegie Mellon Universtiy*, 2009.
[3] L. Yang, J. Cao, S. Tang, T. Li, and A. Chan. A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing. In *Proc. of CLOUD*, 2012.
[4] E. Cuervoy, A. Balasubramanianz, and D. Cho. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. of MobiSys*, 2010.
[5] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. of EuroSys*, 2010.
[6] X. Zhang, A. Kunjithapatham, S. Jeong, S. Gibbs. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. In *Mobile Networks and Applications*, vol.16, no.3, pp.379-394, 2009.
[7] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In *Proceedings of Middleware*, 2009.
[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. In *IEEE Pervasive Computing*, vol. 8, no. 4, pp.14-23, 2009.
[9] K. Kumar, and Y. Lu. Cloud computing for mobile users: Can offloading computation save energy. In *IEEE Computer*, vol. 43, no. 4, pp.51-56, 2010.
[10] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proc. of ICCASES*, 2001.
[11] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. of MobiSys*, 2011.
[12] D.G. Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision*, vol.60, no.2, pp.91-110, 2004.
[13] U. Sharma, P. shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proc. of ICDCS*, 2011.
[14] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. In *IEEE Transactions on Parallel and Distributed Systems*, vol.13, no.3, pp.260-273, 2002.
[15] L. Hall, D. Shmoys, and J. Wein. Scheduling to minimize average completion time: offline and on-line algorithms. In *Proc. of SDA*, 1996.
[16] M. Pinedo. Scheduling Theory, Algorithms, and Systems, 2nd ed. Prentice Hall, Upper Saddle River, New Jersey 07458, 2002.
[17] E.L.Lawler, and D.E.Wood. Branch-and-Bound Methods: A Survey. In *Operations Research*, vol.14, pp.699-719, 1966.
[18] Guido Urdaneta, Guillaume Pierre, Maarten van Steen. Wikipedia workload analysis for decentralized hosting. In *Elsevier Computer Networks*, vol.53, no.11, pp.1830-1845, 2009.
[19] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. ThinkAir: Dynamic resource allocation and parallel execution in cloud for mobile code offloading. In *Prof. of Infocom*, 2012.
[20] Y. Lee, and A. Zomaya. Energy conscious scheduling for distributed computing systems under different operating conditions. In *IEEE Transactions on Parallel and Distributed Systems*, vol.22, no.8, pp.1374-1381, 2011.
[21] Y. Lee, and A. Zomaya. A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems. In *IEEE Transactions on Parallel and Distributed Systems*, vol.19, no.9, pp.1215-1223, 2008.
[22] Y. Kwok, and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, vol.7, no.5, pp.506-521, 1996.
[23] S. Darbha, and D.P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. In *IEEE Transactions on Parallel and Distributed Systems*, vol.9, no.1, pp.87-95, 1998.