Run Time Application Repartitioning in Dynamic Mobile Cloud Environments

Lei Yang, Jiannong Cao, Di Han, Shaojie Tang, and Neeraj Suri

Abstract—As mobile computing increasingly interacts with the cloud, a number of approaches, e.g., MAUI and CloneCloud, have been proposed, aiming to offload parts of the mobile application execution to the cloud. To achieve a good performance by using these approaches, they particularly focus on the *application partitioning problem*, i.e., to decide which parts of an application should be offloaded to the cloud and which parts should be executed on mobile devices such that the execution cost is minimized. Most works on this problem assume that the offloading cost of each part of the application remains the same as the application is running. Unfortunately, this assumption does not hold in dynamic mobile cloud environments, where the device and network connection status may fluctuate, and thus affects the offloading cost. With the varying offloading cost, the one time partitioning of the application may yield significant performance degradations. In this paper, we study *application repartitioning problem* which considers updating the partition periodically during the course of application execution. We first propose a framework for run time application repartitioning in dynamic mobile cloud environments. Based on this framework, we take the dynamic network connection to clouds as a case study, and design an online solution, Foreseer, to solve the mobile cloud application repartitioning problem. We evaluate our solution based on real world data traces that are collected in a campus WiFi hotspot testbed. The result shows that our method can achieve significantly shorter completion time over previous approaches.

Index Terms—mobile cloud computing; application partitioning; repartitioning; trajectory matching

1 Introduction

In recent years, the proliferation of sensors on mobile devices enable a couple of new advanced mobile applications such as augmented reality, collaborative learning, multimedia recognition and retrieval, mobile social gaming and so on. The applications often require intensive and continuous processing of the sensory data. Although the hardware's computing capability increases a lot, running the applications on mobile devices still face problems arising from the constraint on computing capability and/or battery of the device.

On the other hand, the ubiquity and increasing bandwidth of wireless access available to mobile users, and the richness of cloud infrastructures, provide opportunities to develop mobile applications using cloud computing technologies. The most efficient technique used to solve the computing constraints on mobile devices is to offload computations from the device side to the cloud side [1] [2] [3] [4]. By using computation offloading technique, we need to solve the *computation partitioning* problem, which is to partition the application execution between the device side and the cloud side, such that the execution cost such as the latency is minimized. The partition of the application usually depends on the execution environment including the network connection status and the device status.

The computation partitioning problem has been extensively studied in previous research [5] [6] [7] [8] [9] [10] [11] [12] [13]. These works assume the stable/static execution environments during the life cycle of the application, and thus perform one time partitioning according to the execution environment when the application is initiated. The life cycle is defined as the execution time of the application that lasts from the start to termination. However, this assumption does not hold in reality. For instance, the network connectivity can fail when there is no wireless signal or the signal is too weak to maintain a connection. Even when the network is connected, the bandwidth can fluctuate because of user's mobility. Besides the network status, the device status such as the CPU load may vary during the course of the application execution. With the varying network and device status, one time computation partitioning may yield performance degradation. *To motivate this work, in the following, we present one scenario that people run expensive applications while they move, and show the environment, e.g., the network bandwidth, indeed changes a lot during the mobile application's execution.*

An application scenario. One application that is expensive to run on mobile devices while the users move is Augmented Reality (AR), which aims to recognize the scene in reality from the video streams captured

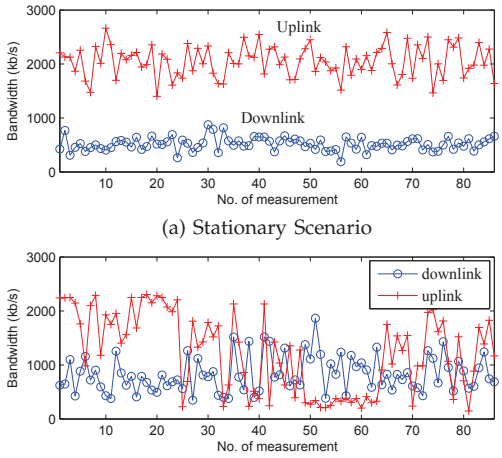
- *Lei Yang is with School of Remote Sensing and Information Engineering, Wuhan University, China. E-mail: yanglei2914@gmail.com*
- *Jiannong Cao is with Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail: csjcao@comp.polyu.edu.hk*
- *Di Han is with Department of Information Technology, Macau University of Science and Technology, Macau. E-mail: andyham.g@gmail.com*
- *Shaojie Tang is with Department of Computer Science, University of Texas at Dallas, USA. E-mail: tangshaojie@gmail.com*
- *Neeraj Suri is with Department of Computer Science, Technical University Darmstadt, Germany. E-mail: suri@cs.tu-darmstadt.de*

Fig. 1. An Augmented Reality (AR) application scenario in the campus environment

by cameras, and then add relevant information into the streams. Fig. 1 shows an AR application scenario in a campus environment. In the application, the users place the cameras of their smart phones to the buildings when they walk around the campus. The application can recognize the building in the campus, and display interesting things in real time on the video streams, e.g., the activities happening in the building. The application can help visitors who are not familiar with the campus to easily find their interested places and activities. The core function in AR is object recognition from the video frames. The device usually executes the function periodically while the user moves, aiming to recognize the varying scene of the surrounding environment in time. We measure the execution time of the recognition function on the main-streaming hardware with 1.7 G Hz 4 Core CPU and 2G RAM. It takes at least 60 seconds to process one 1000*800 frame in the video. If the resolution increases, it can take longer time.

To accelerate the expensive recognition function on mobile devices, existing works like CloneCloud propose to partition the computation between the device and clouds. These works pertain to static partitioning which suffers from low performance in dynamic environments. They made an optimal partitioning every time the application starts to process one frame. During the processing of one frame, the application sticks to the partitioning until that frame is finished. In reality, the mobile device is likely to encounter disconnection, or experience bandwidth fluctuation even when it is connected. For example, if the mobile device gets disconnected or the bandwidth goes down a lot after offloading the computations, the user has to wait for a long time to reconnect to the cloud, in which case a better solution would be executing the application locally. Another example is when current network bandwidth is very low or it is disconnected, CloneCloud would decide to run the application completely on the mobile device. It is possible that the network bandwidth becomes very good shortly after the decision. Therefore, we need to update the partitioning from time to time during the processing of one frame adaptively, based on the changing environment like network bandwidth.

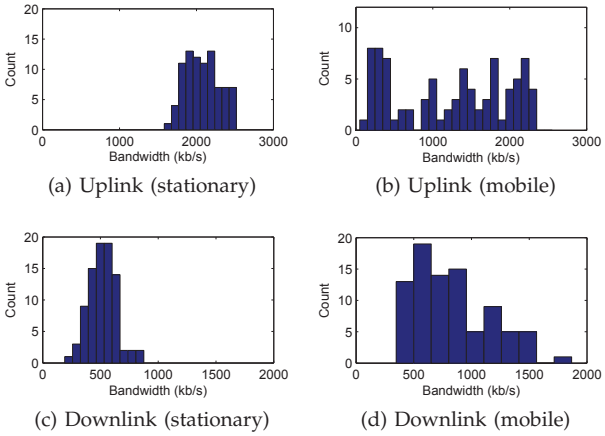
Network bandwidth fluctuation. To learn how the network status fluctuates with time, we have conducted



(a) Stationary Scenario

(b) Mobile Scenario

Fig. 2. Network bandwidth fluctuation in temporal and spatial domain



(a) Uplink (stationary) (b) Uplink (mobile)

(c) Downlink (stationary) (d) Downlink (mobile)

Fig. 3. Histogram of network bandwidth distribution respectively in stationary and mobile scenarios

a series of measurements in a campus WiFi testbed with 23 APs deployed. We compare the network fluctuation between the stationary scenario and mobile scenario. Both scenarios have 86 times of measurements. The measurements are recorded every 20 seconds. Note that in mobile scenario we intentionally avoid the network 'holes' where there exists no wireless signals. Fig.2 shows that in stationary scenario the network status is relatively stable with time, while in spatial domain networks becomes more fluctuant as the user's location changes. The distribution histograms of the measurements shown in Fig.3 as well illustrates the difference. It is shown that the variance of spatial fluctuation is much larger than that of temporal fluctuation. From the measurement above, we conclude that the network bandwidth changes frequently during the application's life cycle, e.g., tens of seconds for the AR application, when the user is moving.

In this paper, we propose the technique of *computation repartitioning* in the dynamic mobile cloud environment, where the network connection to the cloud and device status can vary with time. Computation repartition-

ing updates the partition of application from time to time during its life cycle, according to the estimation and/or the prediction of parameters of the execution environment including the network connection status and device status. More specifically, we first design a framework for run time computation repartitioning in dynamic mobile cloud environments. The framework provides models and mechanisms to conduct the repartitioning of application at the run time.

Based on this framework, we conduct a case study for the computation repartitioning, where the network connection to the cloud fluctuate frequently while the device status is relatively stable. In the case study, we develop an online computation repartitioning method, named as Foreseer. It exploits the knowledge of user's mobility pattern to predict the network status, and then updates the partitioning based on the prediction. We evaluate Foreseer using the data traces that are collected from our campus WiFi testbed. We compare Foreseer against the approach in CloneCloud [6]. It is shown that Foreseer has 35% better performance in term of the completion time. In the case of more frequent network fluctuations, e.g. walking faster in the campus WiFi environment, Foreseer can perform more better than CloneCloud. In summary, our contributions in this paper are three folds.

First, to the best of our knowledge, we are the first to design a framework for run time computation repartitioning in dynamic mobile cloud environments. The framework provides models and mechanisms to solve the performance degradation issue arising from dynamic network and device status. Second, as a case study, we develop an online method, Foreseer, to solve the computation repartitioning problem under the network status fluctuation. We evaluate our method based on real world data traces from our campus WiFi testbed. The result shows that our method can reduce the application completion time by 35% compared with previous approaches.

2 Related Works

The most related works are computation partitioning in mobile cloud computing. We present these works in term of the issues of computation partitioning such as application modeling, environment parameters estimation, and implementation.

There exist three application models: procedure call model, service invocation model, and dataflow model. The works [5][6][10] pertain to the procedure call mode. A procedure call tree or graph is used to model the structure of the application. The partitioning problem is to decide for each procedure whether it should be offloaded or not. The works [7][8] pertain to the service invocation model. A service invocation graph is used to represent the application. [7] decomposes the application with of a set of 'weblets' that can be executed at either the mobile side or the cloud side. The work [8] builds the partitioning system based on a distributed service computing platform, named as AlfredO [14], which have been traditionally used to decompose and couple Java application into software modules. [11][13] models the applications as dataflow graphs in which each node is the stage, and each edge indicates the data dependence between the two connecting stages.

Environment parameters estimation includes the estimation of network status and device status. MAUI [5] conducts an online estimation of the network parameters such as bandwidth and latency through the recent offloading opportunities. It updates the estimation by transferring one 10K file to the server. Frequency transmission of test files incurs a lot of energy overhead on the device. CloneCloud [6] calculates optimal partitions of application under various execution conditions (including the device and network status) in offline phase. The partitions and corresponding execution conditions are stored at the database on the device. In online phase, the system estimates the execution condition, and search the matching partition from the database. It does not discuss how to estimate the execution condition in online phase. [7][10][11] directly estimate the running time of each stage and data transmission time between the stages. This approach avoids the overhead of estimation for the network and device status.

In terms of implementation of mobile-cloud computation partitioning, there exists three approaches, client-server communication, VMs migration, and mobile agent. [3][12] use the client-server communication method to implement the partitioned execution. The method requires the pre-installation of the program codes on the cloud servers. Scavenger [15] uses mobile agent to implement the remote execution. Dynamic deployment of application is realized in this approach. However, it needs agent management that causes overhead on the mobile devices. [5][6][16][10][17] implement the partitioned execution by Virtual Machine migration. The method does not require pre-installation of application on the cloud side.

All the related works [5][6][7][8][9][10][11][12][13] on computation partitioning assumes stable mobile cloud environments, where the device status and network status remains stable during the life cycle of the application. This assumption is reasonable only for the applications that have short life cycle. However, for most applications that have relatively long life cycle, we need to study the repartitioning of application based on the varying device and network status.

3 Terminologies and Application Repartitioning Framework

In this section, we describe the terminologies and our proposed framework for run time computation repartitioning in dynamic mobile cloud environments.

3.1 Terminologies

• *Partition*, *Optimal Partition* and *Compuation Partitioning*. Fig.4 shows the architectural model for the mobile cloud

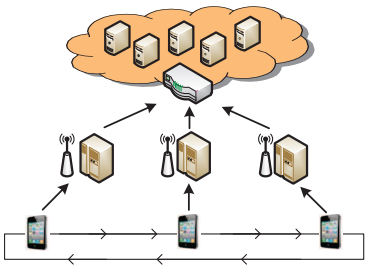


Fig. 4. Architectural model of mobile cloud systems

system. It contains three parts, mobile devices, wireless access network and clouds. The mobile devices can offload some computations of the application to the cloud. Obviously, offloading can reduce the computational cost (e.g., execution time or energy consumption) on the mobile device. Meanwhile, offloading causes additional overhead in data transmissions that are required by the remote execution on clouds. Therefore, in order to minimize the execution cost such as the overall executions latency, it is critical to solve the offloading problem, i.e., to decide whether the application should be offloaded to the cloud or not. For some complex applications which can be divided into a set of dependable parts, we need to make offloading decisions for every part of the application. Note that the decisions for each part are dependent with each other. We name the offloading decisions for all the parts as a *partition* of application. The partition that leads to the minimum execution cost is named as *optimal partition*. The optimization of compuation partitioning according to the network and device status is named as *compuation partitioning*. Computation partitioning changes the execution model of mobile applications, from single machine execution on the mobile device to distributed execution over the device and the cloud.

• *Application Life Cycle* and *Run Time Computation Repartitioning*. *Application life cycle* is defined as the period that the execution of application spans. We also name the period as *run time*. In previous works on compuation partitioning, when the application starts, an optimal partition of the application is determined based on the network and device status at the start time. The partition remains the same during the whole life cycle. *Run time computation repartitioning* is defined as periodically updating of partition of application during its life cycle, based on the changing network and device status, with the aim to reduce the execution cost. Fig.5 illustrates the difference between compuation partitioning and computation repartitioning. The strip with various color represents different partitions. The length of strip indicates the execution time. In compuation partitioning, the application execution sticks to one partition during its life cycle, while in computation repartitioning, the application runs with different partitions.

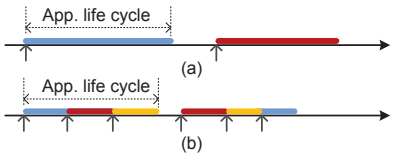


Fig. 5. Illustration: a) computation partitioning; b) computation repartitioning

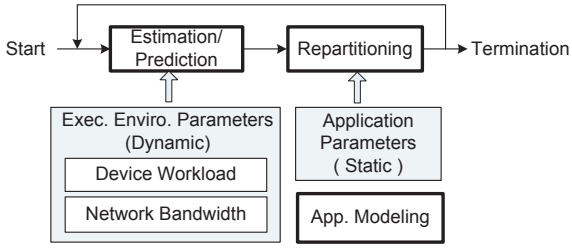


Fig. 6. Functional components for computation repartitioning

3.2 Computation Repartitioning Framework

We now describe the computation repartitioning framework as shown in Fig.6. When the the application is initiated, the prediction component is activated to predict the execution environment parameters, including the device workload and network bandwidth. The repartitioning component outputs a partition based on the predicted environment parameters, and its current execution state. The prediction and repartitioning is alternatively performed during the application life cycle, until the application terminates. The objective of periodical repartitioning is to minimize the total execution cost during the application's life cycle. In practical implementations, the repartitioning is usually done at the cloud side to avoid the additional overhead on the mobile devices. The estimation of parameters can be done by calling third-party services. For instance, as shown in Section 4.2, we predict the network bandwidth on the network server deployed in the mobile networks. Our framework provides methodologies for application modeling, dynamic mobile cloud environment abstraction, and formulation of the repartitioning procedure. We describe the details in the following.

3.2.1 Application Modeling

In our framework, we focus on the method level partitioning of an application. We apply the virtual machine migration to realize the remote execution of application[6] [17]. Each method can be migrated to the cloud. During the migration procedure, the system first captures the runtime state at the mobile device, and then transfers the state to the cloud, and finally reintegrates it back after the execution is finished at the cloud. For simplicity, migration points and reintegration points are restricted to the entry and exit of a method.

We use the method call tree to model the application. In this paper, we interchangeably use the two names,

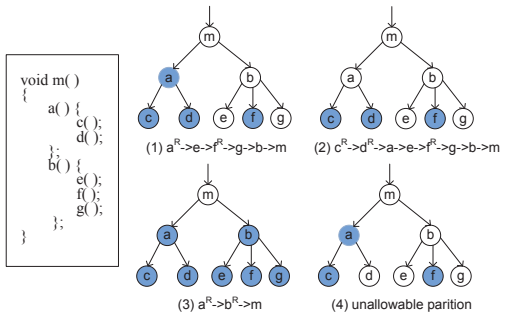


Fig. 7. Program tree, legal partitions and the corresponding execution order.

method call tree and *program tree*. Fig.7 illustrates an example of the program tree. The tree node represents the method and the edge represents the method invocation. For instance, the edge (i, j) indicates that the method i calls the method j. Suppose the methods are executed sequentially between the device and the cloud. Without loss of generality, we require that the program tree is constructed in a way such that the nodes are executed according to the *post-order traversal*. Note that *nested migration* is forbidden. It means that if one node is migrated onto cloud, all its child nodes should be executed at the cloud. Fig.7(1)(2)(3) shows the legal partitions and the corresponding execution order of the methods, where Fig.7(4) shows an unallowable partition, because when the method 'a' is migrated to the cloud, all the children nodes of 'a' should be executed remotely since they are forbidden to migrate back to the mobile side. Note that the colored node means remote execution in cloud, while the others are executed locally. In the execution order, a^R means method a is executed remotely.

We define the following variables for a program tree.

- $C_m(i)$ - The execution cost of method i on the device.
- $C_c(i)$ - The execution cost of method i on the cloud.
- $C'_m(i)$ - The *residual cost* of method i. Normally the cost of the parent node i is larger than the summation of the costs of all its child nodes, because node i contains the cost of running the body of code excluding the costs of the methods called by it. We define *residual cost* for each non-leaf node i by $C'_m(i) = C_m(i) - \sum_{j \in ChildOf(i)} C_m(j)$.
- $S_u(i)$ - The size of VM state that needs to be transmitted to the cloud when the method i is migrated onto the cloud.
- $S_d(i)$ - The size of VM state that needs to be transmitted back to the device when the method i is reintegrated from the cloud to the device.
- $C_s(i)$ - The migration cost of method i. Fig.8 shows that the whole migration procedure contains five phases: suspension, state transfer (uplink), remote computation, state transfer (downlink) and resuming. We assume that the suspension cost C_{susp} and resuming cost C_{resm} are constants for all the methods.

Given the variables above, we can easily obtain the optimal partition, i.e., to decide which methods are executed on the device, and which methods are migrated to the cloud, such that the total execution cost is minimized. The *snapshot partitioning problem* has been well formulated and solved in [5] [6]. In practical systems, the execution cost $C_m(i)$ and migration cost $C_s(i)$ can change during the life cycle of the application in dynamic mobile cloud environments. We need to update the partition accordingly during the application life cycle such that the total execution cost over the life cycle is minimized.

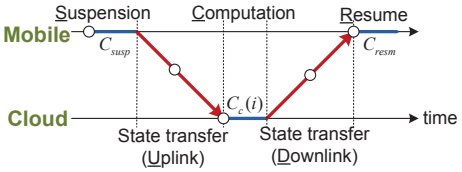


Fig. 8. Execution cost in migration

3.2.2 Dynamic Environment Modeling

We present a simple model of the dynamic mobile cloud environment which includes the device status and network status. The device status affects the execution cost on devices $C_m(i)$. In specific, $C_m(i)$ depends on the computing capability and workload of the device. It is formulated by

$$C_m(i) \propto P \times (1 - \eta),\ 0 \leq \eta < 1, \quad (1)$$

where P denotes the computing capability, i.e., the speed that the device processes the program. η denotes the normalized workloads on the device. It represents the percentages of CPU that have been occupied, where $\eta = 0$ indicates that the CPU is totally idle. The computing capability P is static while the workload η is dynamic.

The network status affects the migration cost $C_s(i)$. In specific, $C_s(i)$ depends on the bandwidth of the network connection to the cloud. If we denote the uplink bandwidth by B_{ul}, and the downlink bandwidth by B_{dl}, $C_s(i)$ is calculated by

$$C_s(i) = S_u(i)/B_{ul} + S_d(i)/B_{dl} + C_c(i) + C_{susp} + C_{resm} \quad (2)$$

In summary, the dynamic parameters of mobile cloud environment that we consider in this paper are device workload η, and network bandwidth (B_{ul}, B_{dl}). In our framework, to guarantee the accuracy, we need to periodically update the estimation and/or prediction of the parameters.

3.2.3 Formulation of Computation Repartitioning

We formulate the repartitioning as a stochastic dynamic decision process. The decision is made every time the execution environment parameters are predicted. We use $n = \{0, 1, 2, ..., N\}$ to denote the decision epoch. The whole life cycle of the application is divided into N decision epoches. We use T_n to represent the length of epoch n. $n = 0$ represents the initial time when the application is launched. The decisions are made at the beginning of every epoch.

System state: The system state is characterized by the joint knowledge of the program *execution status* $\overline{x}_n$ and the *execution environment status* $\overline{y}_n$, i.e., the device workload η and the network bandwidth (B_{ul}, B_{dl}). At each decision epoch n, *execution status* $\overline{x}_n$ reflects the snapshot information of the program. It contains three parameters $\overline{x}_n = (m_n, p_n, \overline{q}_n)$: 1) the method m_n that the program is running at, 2) a binary variable p_n indicating whether it is migrated or not, p_n = 1 if the method is being in migration, otherwise $p_n = 0$, and 3) $\overline{q}_n$ indicating how much computation has been done for this method. If the method runs locally, q_n is assigned with the time that the method has been executed; otherwise if the method is in migration, $\overline{q}_n$ is assigned with: i) the phase that the migration is in, as shown in Fig.8; ii) and the corresponding data size that has been sent to or received from the cloud if the migration is in the phase of state transfer.

We treat $\overline{z}_n = (\overline{x}_n, \overline{y}_n, T_n)$ as the system state at the decision epoch n. We add another state $\overline{z}_n = \mathcal{T}$ to denote the application has been finished. The decision process is terminated when the system enters into state $\mathcal{T}$.

Decision: At each epoch n, after observing the system state $\overline{z}_n$, a decision $\mu_n(\overline{z}_n)$ has to be made. The decision at each epoch n contains: 1) whether or not to terminate the migration procedure if the application is in state of migration; 2)updating the migration points for the methods to be started.

Cost Functions and Optimal Partitioning Policy: The objective of the problem is to minimize the execution time of the application. We define the cost function at each epoch $g(\overline{z}_n) = T_n$ if the program is not completed $\overline{z}_n \neq \mathcal{T}$; otherwise $g(\overline{z}_n) = 0$. The objective function is given by $\rho = \sum_{n=0}^{N} g(\overline{z}_n)$. The sequence of decisions $\phi = \{\mu_0(\overline{z}_0), \mu_1(\overline{z}_1), ..., \mu_N(\overline{z}_N)\}$ is considered as the policy for the dynamic decision process. Let ρ_ϕ be the execution time of the application under the partitioning policy ϕ. The optimal partitioning policy ϕ^* is obtained by $\phi^* = \text{argmin}_\phi \rho_\phi$.

4 Case Study: Compuation Repartitioning under Network Bandwidth Fluctuations

In this section, as a case study, we consider the compuation repartitioning problem under the scenario where the network status encounters connectivity losses and bandwidth fluctuation while the device status is relatively stable.

4.1 Network Fluctuation Model

The network parameter we are concerned about is the user's bandwidth, because it affects the migration cost of the program. Throughout the paper, we exchangeably use the terminologies *bandwidth*, *throughput*, *data transfer rate* and *network status*. We consider a wireless network where users access the network through Base Stations (BSs)/Access Points (APs), and roam from one BS/AP to another to remain connected while they move. There exist some places where the wireless signal is too weak to maintain a connection with the AP, or where there is no signal, because this place is not covered by the BSs/APs or the signal is blocked by surrounding obstacles. We define these places as *'holes'*. Inside holes the user encounters connectivity loss. Outside the holes, the bandwidth can vary as the user moves.

To learn how the network status fluctuates in temporal and spatial domain, we compare the network fluctuation between the stationary scenario and mobile scenario. Fig.2 shows that in stationary scenario the network status is relatively stable with time, while in spatial domain networks become more fluctuant as the user's location changes. The distribution histograms of the measurements shown in Fig.3 as well illustrates the difference. It is shown that the variance of spatial fluctuation is much larger than that of temporal fluctuation. From above measurements, we make an abstraction that the dominating factor that affects the user's bandwidth is the user's geographical location. The reason is that the life cycle of an application usually lasts a few seconds to several minutes at most. If we look at this short period, the bandwidth fluctuation for one user is dominated by the user's mobility.

4.2 Overview of Solution

In Foreseer, we exploit the historical knowledge about the user's mobility to predict the network status. The partitioning of application is then updated based on the predicted network status. In particular, we solve the following problems in the design of Foreseer.

At first, we find that it is extremely hard to accurately measure the network bandwidth in real time through single mobile device. One common approach to measure the bandwidth is by uploading or downloading a large file to or from the server. The uplink or downlink bandwidth will be the size of the file divided by the time used for the uploading or downloading. This measurement itself takes several seconds at least, which is not acceptable for our case. In addition, the high overhead that is incurred during the measurement, e.g., additional data transmission and battery consumption, makes it infeasible to conduct such bandwidth measurement via single mobile device. In Foreseer, we leverage the crowdsourcing to collect the users' bandwidth together with their locations, and learn the probabilistic model of network bandwidth conditioned on the location. The user can query the network bandwidth with its location.

Second, we need to accurately predict the future network bandwidth from the user's mobility. In Foreseer, we deploy a centralized server on the mobile network, named as *network server* throughout the paper, for: (1) providing the service that allows the mobile users to share location-bandwidth pairs; (2) collecting the users' *historical trajectories*; (3) performing online trajectory

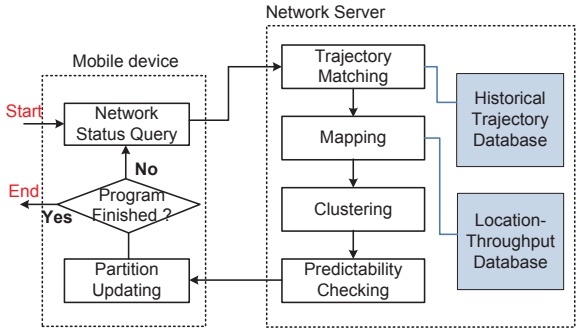


Fig. 9. Flow chart of Foreseer

matching and network status prediction. To improve the prediction accuracy, we only predict the network status up to certain point in the future, which we name as *predictable duration*. The uncertainty of the network status is relatively low in the predictable duration, while beyond the predictable duration the network status becomes highly uncertain. The network status for one particular user can be predicted based on either the historical trajectories by individuals, or the historical trajectories by all the users. We realize that a centralized server may become the bottleneck of the system when the users scale up. The challenge of designing scalable architectures is inherent to all distributed systems and beyond the scope of this paper.

With the predicted network status, the core of our problem is to develop an online decision policy for the dynamic repartitioning process. We design an online algorithm that can work efficiently in real systems. Our solution is to make the decision that maximizes the *execution progress* in the predictable duration. Execution progress is defined as the position of the program counter while the application is in execution. In the following, we present a high level description of the protocol which shows how the components work together in our system. Fig.9 shows the detailed flow chart of Foreseer.

1. Network Status Query. When the mobile user enters the network, it receives the beacons from the surrounding APs periodically. It measures the signal strength of the beacons, and labels its location with a sequence of AP IDs, which are sorted in a descending order of their signal strength. The location label is null when the user is in 'holes'. When the application is launched, the mobile user sends a series of recorded locations to the *network server*, and queries for its network status.

2. Network Status Prediction. With the series of locations, the *network server* searches the historical trajectories from the database whose prefixes match with the query. All the matched trajectories are mapped into sequences of network bandwidth values, based on the location-bandwidth fingerprint database. Usually, the longer future we attempt to predict, the more uncertain the network bandwidth will be. The network bandwidth sequences are cut off at some time in future, called as *predictable duration*, such that the uncertainty about the network status is low enough. The most likely sequence is returned as the result.

3. Partition Initializing/Updating. Upon receiving the network status, the user determines/updates the partitioning decision such that the execution progress is maximized in this predictable duration. Towards the end of the predictable duration, the mobile user will send the network status query again for the next partition updating.

The three steps are performed iteratively until the program is finished. If the user happens to be in 'holes' when it sends the network status query, the query will be postponed until the user moves out of the 'holes'. The application sticks on the previous partition.

4.3 Network Status Prediction

4.3.1 Trajectory Representation

We represent the user location with a set of AP IDs from which the mobile user is able to detect the beacon signal. The APs are sorted by a descending order of the signal strength. For example, a legal representation of location is 'abc', in which case the signal from AP 'a' is the strongest, 'b' is less strong and 'c' is the weakest. If the user is located at the holes, the location is labeled as 'N'. There are two cases that the hole 'N' is labeled. The first case is that no AP signal is detected by the user. Second, the signal from the detected AP is too weak to maintain the connection. We have done experiments to learn the threshold of the signal strength, below which the network connection is not possible. We found a threshold of 90 db as an empirical value. The trajectory is represented as a time series of locations. One example of a trajectory is: "ab ab abc bc c c d d N N N f". In the string, we use the space to isolate two locations.

4.3.2 Trajectory Matching

We do not have any requirement on the length of the recorded trajectory. For convenience of description, we can abstract all the trajectories at the database as one virtual string, by joining them together with special isolated symbols. We name the virtual string as *historical string* or *history*, and the string that the user sends for querying the network status as *contextual string* or *context*. The trajectory matching is to find the positions at which the context occurs in the historical string. The substrings of the history, which occur immediately after the context, will be returned as the possible trajectories that the user will pass by.

In a practical solution, we need to determine a proper length of the context in the string matching. If the context is too long, we may not have enough samples to estimate the probabilistic distribution of the future trajectory. Oppositely, short contexts can not fully capture the feature of the user's input trajectory. In our system, we use the approach in Sampled Pattern Matching (SPM) algorithm [18] to determine the length of the context. In contrast

to k-order Markov predictor, which uses a fixed length of the context, the length of context in SPM is decided by a fixed fraction α of the longest context that occurs in the history, where $0 < \alpha < 1$. The method is appropriate to be used in the prediction based on large diverse data sources (in our case, the trajectory traces are from various users at different times).

4.3.3 Mapping

In our system, we build a location-throughput database on the network server. There exists many efficient ways to construct the fingerprint database[19]. In our system, we use the most intuitive yet efficient method. The database is directly indexed using the location (of string type). For each indexed location, we determine the corresponding throughput simply by a weighted averaging of the samples at that location. The latest samples are given higher weights. The throughput for each indexed location is updated as new samples are added. The Based on the database, we can map all the possible trajectories into sequences of network throughput values. Note that the location labels 'N' for the network holes are directly mapped into zeros.

4.3.4 Clustering

Suppose that we have M sequences and the length of each sequence is N. Let $\bar{b}_i = (b_{i,1}, b_{i,2}, ..., b_{i,N})$, $1 \leq i \leq M$ denote the sequence i. We perform the clustering for the M sequences. The sequences that are similar to each other will be grouped into one cluster. We measure the similarity between two sequences by their Euclidean distance

$$d(\bar{b}_i, \bar{b}_j) = \frac{1}{N}\sqrt{\sum_{k=1}^{N}(b_{i,k} - b_{j,k})^2}. \quad (3)$$

If the distance of two sequences is lower than a given threshold, we say that the two sequences are similar to each other. In the clustering, we construct a graph for all the M sequences, in which the nodes represent individual sequences, the edge represents that the two connected nodes are similar to each other. The clustering is to iteratively find the maximum clique from the graph. A lot of heuristics have been proposed for the Maximum Clique Problem (MCP) [20]. We will not describe the details in our paper. The cluster that has the largest size is selected to represent the predictable result. In our solution, since the sequences within the cluster are highly similar to each other, we randomly choose one from the cluster as the future network throughput sequence.

4.3.5 Predictability Checking

We define *predictability* by the entropy of the network throughput distribution in future time. Suppose we get K clusters after the clustering for the M throughput sequences. Let $\mathcal{C}_k$ denotes the cluster, where $1 \leq k \leq K$, and $S(\mathcal{C}_k)$ denote the size of the cluster, where $\sum_{k=1}^{K} S(\mathcal{C}_k) = M$. The probability that the actual result is from the cluster $\mathcal{C}_k$ is $p(\mathcal{C}_k) = \frac{S(\mathcal{C}_k)}{M}$. With the probabilistic distribution of the clusters, the predictability is defined by the entropy $H = -p(\mathcal{C}_k)\log_2 p(\mathcal{C}_k)$. Entropy reflects the uncertainty of the future network status. Greater entropy indicates the lower predicability. Generally, the longer the network status to be predicted is, the lower the predictability is.

In our solution, we start from a small prediction duration and check the entropy. If the entropy is lower than a threshold H_{th}, we continue to extend the prediction duration in future. The prediction procedure is terminated until the entropy of the network throughput distribution exceeds H_{th}. We name the length of the period as *predictable duration*. In the computation repartitioning problem, the length of the decision epoch is the predictable duration. Note that we distinguish the uplink and downlink in our system, since our real measurements show the two are quite different. In our solution, we treat them as two independent variables. After we obtain the possible trajectories, we predict the uplink throughput and downlink throughput independently. It is possible that the two have different predictable durations. In this case, we choose the smaller value as the predictable duration, without affecting the predictability of the other one.

4.4 Computation Repartitioning

We first describe the solution for the offline partitioning problem, where the future network status is perfectly known. By using our offline solution, we further design an online algorithm for the compuation repartitioning problem.

4.4.1 Offline Algorithm

The offline problem is to determine an optimal partition of the application, assuming that the network status in the future is known, such that the completion time of the application is minimized. The offline partitioning problem is different with snapshot partitioning problem. In snapshot partitioning problem, the migration cost $C_s(i)$ of each method is constant, while in offline partitioning problem the migration cost depends on when the migration happens. Existing works [5] [6] aim to solve the snapshot partitioning problem, but can not solve the offline partitioning problem.

We develop a recursive algorithm for the offline partitioning problem as shown in Algorithm 1. The input variables includes the local execution cost of each node $C_m(i)$, and the migration cost related variables such as C_{susp}, C_{resm}, $C_c(i)$, $S_u(i)$, $S_d(i)$, and the dynamic network uplink and downlink bandwidth, denoted as $B_{ul}(t)$ and $B_{dl}(t)$. Note that the network bandwidth during the time period $(0, \infty)$ is known. The application is launched at some time point t. The algorithm outputs the optimal partition. Let $Y(i)$ represent the partition. $Y(i) = 1$ if node i is migrated onto the cloud, otherwise

Algorithm 1: The offline algorithm

Offline Algorithm: ***FindOptimalPartion***(i, t)

```
t_c ← t;
if i ∈ LeafNodes then
    t_nmgr(i) ← C_m(i);
else
    t_nmgr(i) ← C'_m(i);
while j ∈ ChildrenOf(i) do
    t_nmgr(i) ← t_nmgr(i) + FindOptimalPartion(j, t_c);
    t_c ← t_c + FindOptimalPartion(j, t_c);
if t_nmgr(i) > C_s(i, t_0) then
    Y(i) ← 1; // Method i is migrated into cloud;
    t_opt(i) ← C_s(i, t_0);
else
    Y(i) ← 0; //Method i is executed locally;
    t_opt(i) ← t_nmgr(i);
return t_opt(i);
```

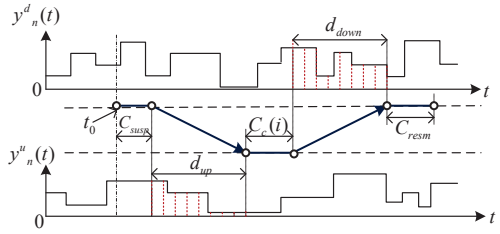


Fig. 10. How to calculate migration cost

$Y(i) = 0$. Algorithm 1 can optimize the partition for any program subtree i that is going to start at time t, and return the corresponding completion time. Since the network bandwidth is changing over time, the migration cost of node i depends on the time it is started. In the algorithm, $C_s(i, t_0)$ represents the migration cost of node i if the migration procedure starts at time t_0. Fig.10 shows the migration cost as given by $C_s(i, t_0) = C_{susp} + d_{up} + C_c(i) + d_{down} + C_{resm}$, where d_{up} and d_{down} are the state transfer time that satisfy:

$$\int_{t_0+C_{susp}}^{t_0+C_{susp}+d_{up}} y^u(t)dt = S_u(i), \tag{4}$$

$$\int_{t_0+C_{susp}+d_{up}+C_c(i)}^{t_0+C_{susp}+d_{up}+C_c(i)+d_{down}} y^d(t)dt = S_d(i). \tag{5}$$

4.4.2 Online Algorithm

We first define application *execution progress* and then describe the online algorithm. *Execution progress* is defined as the position of the program counter while the application is in execution. In particular, if the program is being in migration, execution progress is defined as the position of the migration point, although it is possible that the actual program counter at the cloud side is in advance of the migration point. Note that *execution progress* is different with the *execution status* defined in Section 3. Fig.11 illustrate execution progress given the application *execution status*. Suppose that the application is totally completed at mobile device, we can construct a *progress bar* that shows the start time and end time of

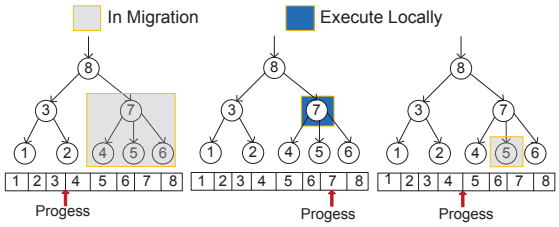


Fig. 11. Execution progress

each method. In this example, the progress bar is in the order of {1, 2, ..., 8}. Note that the length of execution time of the non-leaf node is its *residual cost* $C'_m(i)$, which represents the cost of running the body of code excluding the costs of the methods called by it. In our solution, we simplify that the body of code of the non-leaf node is executed after all its children nodes. After the progress bar is constructed, for any feasible application execution status, we can shows its progress on the bar. The left tree shows that method '7' is being in migration. The execution progress is the migration point of '7', which is right after the completion time of method '3'. The tree in the middle shows that method '7' is being executed on the mobile device. The progress is located at some position of '7' on the bar. In the right tree, method '5' is being in migration, thus the execution progress is at the end of '4' on the bar.

In the online solution, we maximize the application *execution progress* at each decision epoch. According to the definition of execution progress above, if and only if the method in migration can be returned before the end of the current epoch, it will contribute to the execution progress. Oppositely if the method is migrated but not re-integrated in this epoch, the time that is spent on the method migration has no contribution to the execution progress. This is because of our pessimistic estimation about the network status beyond the predictable duration. The worst case that the communication is fully disconnected would happen beyond the epoch. Thus, we have a conservative migration policy in our online solution: if the method is able to re-integrate back to the device in the epoch, the migration of the method is allowed; otherwise the migration is not allowed, because in this case we can always obtain more progress by executing the method locally. We conclude that the problem of finding the partition that maximizes execution progress at current decision epoch is equivalent to the problem of finding the partition, that minimizes the completion time of the application, given that the network throughput beyond the predictable duration is zero.

Algorithm 2 shows the online algorithm of the compuation repartitioning problem. The algorithm first checks the execution state $\overline{x}_n = (m_n, p_n, \overline{q}_n)$. If the method m_n is being in migration procedure, the algorithm makes the decision X_n of whether to terminate the migration or not, by comparing the time t_{cont} that the subtree m_n needs to be finished if the migration continues, and the

Algorithm 2: The online algorithm

Online Algorithm: $\mu_n(z_n)$
1 **if** *The program is being in migration* $p_n = 1$ **then**
2 $t_{cont} \leftarrow EstimateFinishTimeOf(m_n)$;
3 $t_{term} \leftarrow C_{resm} +$ ***FindOptimalPartition***(m_n, C_{resm});
4 **if** $t_{cont} < t_{term}$ **then**
5 $X_n \leftarrow 0$; $t \leftarrow t_{cont}$;
6 **else**
7 $X_n \leftarrow 1$; $t \leftarrow t_{term}$;
8 **else**
9 $t = EstimateFinishTimeOf(m_n)$;
10 $i \leftarrow m_n$;
11 **while** $j \leftarrow NextNodeOf(i)$ **do**
12 **if** $j = ParentOf(i)$ **then**
13 $t = t + C'_m(j)$;
14 **else**
15 ***FindOptimalPartition***(j, t);
16 $i \leftarrow j$;

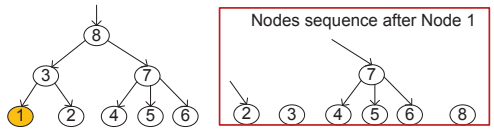


Fig. 12. Nodes sequence for $NextNodeOf()$

time t_{term} if m_n terminates the migration and seeks a different partition. Note that the overhead of switching to different partition is C_{resm}. t_{cont} can be estimated based on the application *execution status* and network status. As the network throughput beyond the this epoch is assigned with zero, if the migration is still not finished in this epoch, we have $t_{cont} = \infty$. Line 13 to 20 is to update the partition for the nodes not started. When we search the next node to be started, we always search the subtree that includes as more children that are not started as possible. Fig.12, for example, shows the next node after 1 is nodes 2, 3, 7, 8 rather than 2, 3, 4, 5, 6, 7, 8.

The predicted network status is not ideally the same with the reality, so the program is likely to end up at each epoch with a method being in progress of migration. In this case, the decision on whether or not to terminate the migration procedure is necessary. At each epoch, we make the partition for all the nodes to be started rather than the nodes that are likely to be started in current epoch. It causes more overhead but it is reliable to handle the case in which the network throughput happens to be much better than what we predicted, and some nodes that were not considered to start in the epoch start eventually.

5 Evaluation

5.1 Evaluation Setup

We collect the network bandwidth traces from our campus WiFi network testbed. We deploy 23 WiFi access points in the test area. Note that in the Fig.13 only the nodes labeled with figures are deployed with APs. The 23 APs are densely deployed at the four buildings. Some

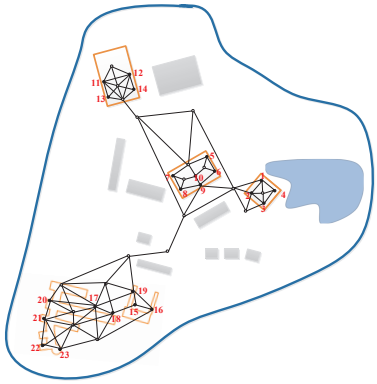
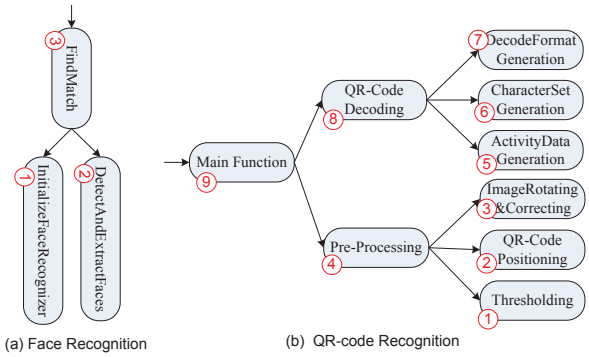


Fig. 13. APs deployment and Mobility Graph.

buildings with APs deployed start from the first floor, so people can freely pass through them on the ground. The maximum length and width of the erea are approximately 500 meters and 700 meters. The APs are mainly deployed at four buildings. Since APs are not intentionally deployed to cover the whole test area, there exists a few network holes both in the buildings and in the open space of the campus. Our prediction method needs the database of historical trajectories. Before collecting the trajectories, we have built a *mobility graph* that constrains the user's mobility due to the environment restrictions. The users' trajectories correspond to pathes in the graph. Fig.13 shows the AP deployment and mobility graph in our test area. We have collected data for 30 trajectories for six users. The trajectories are not totally different, but have a few overlaps between each other. The locations are recorded every two seconds along the trajectory, but each trajectory has different speed that ranges from approximately 1.0 m/s to 2.5 m/s. The time length of each trajectory is about 10 minutes.

In order to build a snapshot map of location-throughput, we have selected 200 positions to measure the network bandwidth which covers the whole test area. At each position we have ten measurements of both the downlink and uplink bandwidth. The average on the 10 measurements is recorded as network status at that position. The measurements are collected from 10:00 am to 12:00 am on Mar 2nd, 2013. Note that the positions we selected do not include network holes, because the network bandwidth in holes can be directly mapped into zero. In real systems, the snapshot map of location/throughput varies depending on workloads on APs, traffic on the backbone and so on. It needs to be updated periodically, e.g., one update per hour. In our evaluation, we simply collect the snapshot map once and use it all the time.

We evaluate our repartitioning algorithm using two applications: face recognition and QR-code recognition. These two applications are also used in most related works [5] [6] [13]. Fig.14 shows the method level graph of the two applications. The details about the application parameters are shown in Table 1, where the data of face recognition are from [5] and the data of QR-code



(a) Face Recognition (b) QR-code Recognition

Fig. 14. Program trees used in evaluation

TABLE 1
Application parameters

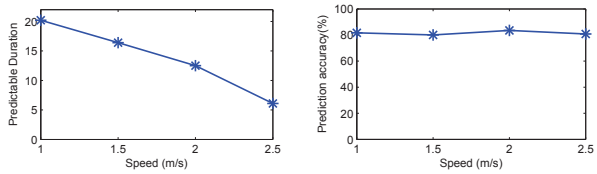
App.	Meth.	$C_m(i)$	$C'_m(i)$	$C_c(i)$	$S_u(i)$	$S_d(i)$
Face Recog.	①	7.1 s	7.1 s	0.71 s	2.3MB	2.3MB
	②	19.6 s	19.6 s	1.96 s	6.9MB	6.9MB
	③	28.1 s	1.4 s	0.14 s	91kB	91kB
QR-Code Recog.	①	800ms	800ms	80ms	350KB	87.5KB
	②	1300ms	1300ms	130ms	87.5KB	62.5KB
	③	110ms	110ms	11ms	62.5KB	87.5KB
	④	2210ms	0 ms	221ms	350KB	87.5KB
	⑤	500ms	500ms	50 ms	62.5KB	50.3KB
	⑥	400ms	400ms	300 ms	62.5KB	50.1KB
	⑦	300ms	300ms	30 ms	62.5KB	50.1KB
	⑧	1480ms	280ms	148ms	87.5KB	63KB
	⑨	3690ms	0 ms	369ms	350KB	52KB

recognition are from [13]. Note that the computation in cloud is 10 times faster than on mobile devices. The suspension cost C_{susp} and resuming cost C_{resm} are set as 1 second. These are typical values used in [6].

5.2 Network Status Prediction

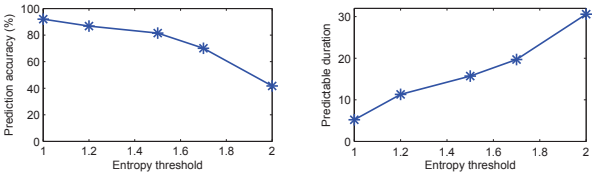
We first evaluate our approach for network status prediction. We concern on two metrics: *accuracy* and *predictable duration*. The accuracy is measured by the number of successful prediction over the total number of predictions. *Successful prediction* means that the predicted result is similar to the ground truth according to Equation 3. The predicted result is obtained through mapping the predicted trajectory into network bandwidth sequence by using the location-bandwidth map. However, to obtain the ground truth, we first map the real trajectory into network status sequence. Considering the stochastic property of bandwidth in temporal domain (shown in Fig.2a), the ground truth are then added with a simulated Gaussian noise. The mean and variance of the noise added for uplink and downlink bandwidth are respectively set as $(0, 400kbps)$ and $(0, 100kbps)$.

We evaluate the overall performance of the prediction method. We choose one of the 30 trajectories as test trajectory, and the left as the historical trajectories. We simulate the online predictions with the test trajectory.



(a) Prediction accuracy (b) Predictable duration

Fig. 15. Performance of network status prediction varies depending on the walking speed of user



(a) Prediction accuracy (b) Predictable duration

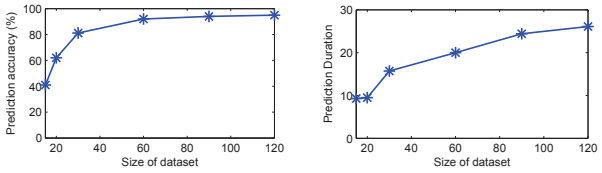
Fig. 16. Performance of network status prediction varies depending on H_{th}

We repeat this evaluation on all 30 trajectories. Finally, we have 1123 times of prediction, where each trajectory has $1123/30 = 37$ times of predictions on average. Among those predictions, 912 predictions are successful, thus the overall accuracy of our method is 81.2%. The average predictable duration is 15.7 seconds. The result shows we can accurately predict the network status in future 15.7 seconds with our prediction method.

Next, we evaluate how the walking speed affects the prediction performance. We classify the trajectories into four categories according to the speed: slow (about 1m/s), medium (about 1.5m/s), fast (about 2.0m/s), and very fast (about 2.5 m/s). For the test trajectories which belong to the same speed category, we count the the total number of online predictions, the number of successful predictions, and the average predictable duration. Fig.15 shows predictable duration decreases as the speed increases, while the accuracy almost remains the same. The reason that the predictable duration changes is, that the length of future spatial trajectory that we are able to predict is constrained. If the users move fast, the time that user passes by the predictable area will be short. The speed does not affect the accuracy because of the threshold H_{th} that is applied to stop prediction if the network status becomes high uncertain in further future.

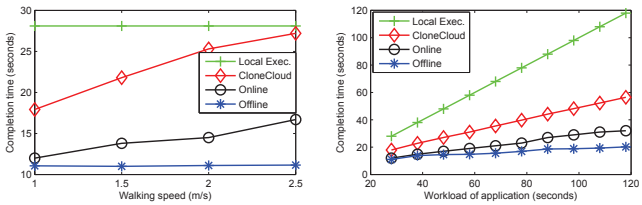
We further evaluate the effect of parameter H_{th} on the prediction accuracy. Increasing H_{th} means allowing prediction in uncertain network status distribution. In this case, although the predictable duration could be increased, the prediction is more likely to deviate from the ground truth. Fig.16 shows prediction accuracy decreases as we increase the threshold H_{th}. We will report how H_{th} influences the performance of program partitioning in next subsection.

Finally, we explore the impact of the trajectory data



(a) Prediction accuracy (b) Predictable duration

Fig. 17. Performance of network status prediction varies depending on the data size



(a) walking speed (b) application workload

Fig. 19. The completion time varies depending on: a)the walking speed; b) application workload

size on the prediction performance. First, we discard some of the 30 trajectories data set. As shown in Fig.17, we find both the accuracy and predictable duration decrease significantly. The reason is that the test trajectory may have some spatial intervals that never appear in the historical trajectory database. In this case, it is difficult to find the matched trajectory from the database, or even when matched trajectories are searched from the database, but the successive part of the test trajectory appears to be different from all the matched trajectories. Second, we increase the data size by simulating more trajectories. The new trajectories are generated by randomly picking up the intervals from the 30 real trajectories and joining them together. Fig.17 shows that both prediction accuracy and predictable duration increases as the data size increases. In particular, the increase of predictable duration is very obvious. Overall, this evaluation implies that the performance of our prediction method highly relies on the data size. If we want to be able to predict more time in future or more accurately, we should have more samples in the historical database.

5.3 Computation repartitioning

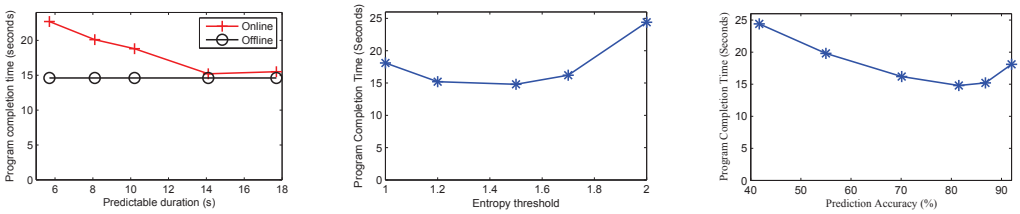
Metric: Completion Time. We compare the online compuation repartitioning method (Algorithm 2) respectively with CloneCloud, which runs the application with one time partitioning based on the current network status when the application is launched, the offline partitioning method (Algorithm 1) and the baseline case that the application is totally executed on the mobile device. The main performance metric is *completion time* of the program. The performance of the offline partitioning algorithm actually reflects the upbound that the online algorithm can achieve. The algorithms are evaluated using the network traces we have collected.

First, we evaluate the overall performance of the partitioning methods. We have collected 30 network traces. For each network trace, we repeatedly run the application 50 times by randomly selecting the application launching time along the trajectory. Thus, the application runs $50 \times 30 = 1500$ times for each method. We obtain the average completion time for the online algorithm, CloneCloud, offline algorithm and the baseline method. Fig.20a and Fig.20b respectively show the program completion time under the four methods for the two applications. For face recognition, Foreseer can reduce the completion time by 41% over CloneCloud. For QR-code recognition, Foreseer can reduce the completion time by 35% over CloneCloud.

We analyze how predictable duration affects the online predictive method. Note that we run the program for 1500 times to obtain the overall performance of the predictive method. Each running of the program requires multiple times of prediction on the network status. We record the average predictable duration as well the program completion time during each running of the program. We analyze the 1500 samples of the predictable duration and program completion time, and find that program completion time decreases as the predictable duration increase, which is shown in Fig.18a. *This result implies that if we can predict more time in future about the network status, we will achieve better performance.*

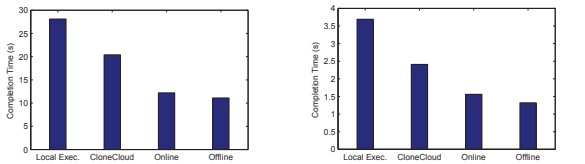
To evaluate how the performance of predictive partitioning algorithm changes depending on the prediction parameter H_{th}, we assign H_{th} with different values, and repeat 1500 runs of the algorithm. Fig.18b shows that the performance is not good either when H_{th} is too low or H_{th} is too high. The reason is that when H_{th} is too low, although the network status can be predicted accurately (shown in Fig.16a), the predictable duration is quite short which degrades the performance. Oppositely when H_{th} is too large, the network prediction is not accurate, which leads to bad performance of the predictive method. In addition, we record the network bandwidth prediction accuracy under each H_{th} in this evaluation. Fig.18c plots the sensitivity of the online partitioning approach to the prediction accuracy of the network bandwidth.

We then evaluate how the walking speed affects the performance of the three partitioning methods. We conduct the test under four speeds 0.5 m/s, 1.0 m/s, 1.5 m/s, and 2.0 m/s. For each speed, we select one trajectory, and also repeat the application 50 times by randomly choosing the application launching time. Fig.19a shows the three methods' performance under different walking speeds. The performance of both the online method and CloneCloud degrades as the walking speed increases. The faster the user walks, the more unstable the networks status is, in which case CloneCloud that assumes the stable network has lower performance. The reason that our online algorithm has lower performance as the speed increases is that the predictable duration becomes short when user moves faster. *This evaluation shows that*



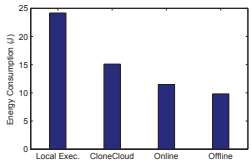
(a) (b) (c)

Fig. 18. The completion time varies depending on: a)the predictable duration; b) H_{th}; c) the prediction accuracy

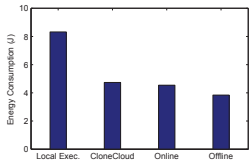


(a) FR - Completion Time

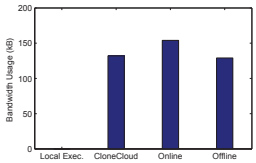
(b) QR - Program Completion Time



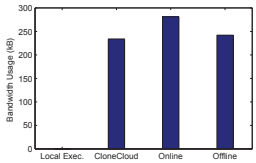
(c) FR - Energy Consumption



(d) QR - Energy Consumption



(e) FR - Bandwidth Usage



(f) QR - Bandwidth Usage

Fig. 20. Performance comparison between four methods: **CloneCloud**, Foreseer (**Online** Algorithm), Foreseer (**Offline** Algorithm) and **Local Execution** without partitioning.

Foreseer always outperforms Clone Cloud, specially when the user moves relatively fast in the network.

We evaluate the effect of application workload on performance of the partitioning methods. The *workload* is defined as the completion time if the program is totally executed on the mobile device. In this evaluation, we still use the face application, but we simulate large workload by assuming that the same application is executed on a very slow device. Fig.19b shows the completion time of the three methods vary depending on the program workload. We can see that the more the program workload is, the better performance our predictive online algorithm has over CloneCloud. *This evaluation implies that Foreseer is suitable to be used in compute-intensive applications, while for small application, it is enough to use current bandwidth to partition the program.*

Other Metrics: Energy Consumption and Bandwidth Usage. Although Foreseer is designed with the objective of minimizing program completion time, we also measure other metrics of Foreseer such as energy consumption and bandwidth usage. The *energy consumption* of Foreseer contains the application execution itself and overhead of network status prediction. To measure the energy consumption of application itself, we use the same energy model with [6] in our evaluation. For the overhead of Foreseer, we mainly consider the energy consumed on the periodic location sensing. Therefore, we have the following equation to model the energy consumption of Foreseer: $EC = P_{cpu}t_{comp} + P_{net}t_{trans} + E_{sense}$. P_{cpu} and P_{net} are constants. E_{sense} is calculated by the total energy consumed in location sensing among the whole trajectory divided by the run times of the program on this trajectory. The *bandwidth usage* indicates the data amount that are transferred over the wireless networks. This metric is usually concerned by the users who have limited data traffic budget.

Fig.20 shows the energy consumption and bandwidth usage for the two applications. All measurements are the average of 1500 runs of the application. We can see that Foreseer outperforms CloneCloud in term of energy consumption for the application of face recognition, while for QR-code recognition, Foreseer consumes the same amount of energy with CloneCloud. This is because face recognition has much larger workload then QR-code recognition. Usually Foreseer can save energy through reducing the completion time. *relatively large workload programs can benefit from the Foreseer in term of energy consumption.* The bandwidth usage for CloneCloud and Foreseer(offline) are almost the same for the two applications. However, *Foreseer (online) causes a litter bit higher bandwidth usage than the other two schemes.* This is because the online algorithm of Foreseer can terminate the method migration if it predicts that future network bandwidth is not good. In this sense, the bandwidth used to transmitting data in the terminated migration procedure is a waster.

6 Conclusion

In this paper, we proposed a framework for run time compuation repartitioning in dynamic mobile environments. Based on the framework, we take the dynamic network connection to the cloud as a case study, and design an online solution for compuation repartitioning under network fluctuations. Our solution exploits

the knowledge of user's mobility to predict the future network status. According to the network prediction, we designed the online repartitioning algorithm that aims to maximize the execution progress during current the predictable duration. To evaluate our solution, we collected data set from our campus Wifi testbed that contains the user's walking trajectories and measurements of spatial distribution of network throughput. The evaluation results show that our solution can reduce the completion time of program by at least 35%.

ACKNOWLEDGMENTS

The research is partially supported by Hong Kong RGC under GRF Grant 510412, Microsoft under Grant H-ZD92, and the National High-Technology Research and Development Program (863 Program) of China under Grant 2013AA01A212.

REFERENCES

[1] K. Kumar and Y. Lu, "Cloud computing for mobile users: Can offloading computation save energy," *Computer*, vol. 43, no. 4, pp. 51–56, 2008.
[2] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi, "Using bandwidth data to make computation offloading decisions," in *Proc. of IPDPS*, 2008, pp. 1–8.
[3] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proc. of ICDCS*, 2002, pp. 1–10.
[4] M. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *Proc. of INFOCOM*, 2013, pp. 1285–1293.
[5] E. Cuervoy, A. Balasubramanianz, and D. Cho, "Maui: Making smartphones last longer with code offload," in *Proc. of MobiSys*. ACM Press, 2010, pp. 277–289.
[6] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. of EuroSys*, 2011, pp. 301–314.
[7] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs, "Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing," *Mobile Networks and Applications*, vol. 16, no. 3, pp. 379–394, 2009.
[8] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: Enabling mobile phones as interfaces to cloud applications," in *Proc. of Middleware*. ACM Press, 2009, pp. 1–20.
[9] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: a partition scheme," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.
[10] 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 *Proc. of INFOCOM*, 2012, pp. 945–953.
[11] M. Ra, A. Sheth, L. Mummert, P. Pillai, and D. Wetherall, "Odessa: enabling interactive perception applications on mobile devices," in *Proc. of MobiSys*. ACM Press, 2011, pp. 43–56.
[12] R. Balan, M. Satyanarayanan, S. Park, and T. Okoshi, "Tactics based remote execution for mobile computing," in *Proc. of MobiSys*. ACM Press, 2003, pp. 945–953.
[13] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SigMetrics Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.
[14] J. Rellermeyer, O. Riva, and G. Alonso, "Alfredo: An architecture for flexible interaction with electronic devices," in *Proc. of Middleware*. ACM Press, 2008, pp. 22–41.
[15] M. Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *Proc. of PerCom*, 2009, pp. 217–226.
[16] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 6, no. 4, pp. 12–23, 2009.
[17] M. Gordon, D. Jamshidi, S. Mahlke, Z. Mao, and X. Chen, "Comet-code offload by migrating execution transparently," in *Proc. of OSDI*, 2012, pp. 93–106.
[18] P. Jacquet, W. Szpankowski, and I. Apostol, "An universal predictor based on pattern matching, preliminary results," *Mathematics and Computer Science: Algorithms, Trees, Combinatorics and Probabilities*, pp. 75–85, 2000.
[19] V. Hoonkavirta, T. Perala, S. Ali-Loytty, and R. Piche, "A comparative survey of wlan location fingerprinting methods," in *Proc. of WPNC*, 2009, pp. 243–251.
[20] M. Bomze and M. Budinich, *Handbook of Combinatorial Optimization*. Springer, 1999.

Lei Yang received his Ph.D degree from Department of Computing, Hong Kong Polytechnic University, in 2014, the MSc degree from Institute of Computing Technology, Chinese Academy of Science, in 2010, and the BSc degree from Wuhan University, in 2007. He is currently an assistant professor from School of Remote Sensing and Information Engineering in Wuhan University. His research interest includes mobile cloud computing, RFID systems, and social computing.

Jiannong Cao is currently a chair professor and the head of the Department of Computing at Hong Kong Polytechnic University. He received the BSc degree from Nanjing University, China, in 1982, and the MSc and PhD degrees from Washington State University, USA, in 1986 and 1990, all in computer science. His research interests include parallel and distributed computing, computer networks, mobile and pervasive computing, fault tolerance, and middle-ware.

Shaojie Tang received his Ph.D degree from Department of Computer Science at Illinois Institute of Technology in 2012. He received B.S. in Radio Engineering from Southeast University, P.R. China in 2006. He is a member of IEEE. His main research interests focus on wireless networks (including sensor networks and cognitive radio networks), social networks, pervasive computing, mobile cloud computing and algorithm analysis and design.

Di Han is currently a Ph.D student from Department of Information Technology in Macau University of Science and Technology. He worked as a research assistant of the Department of Computing at Hong Kong Polytechnic University in 2012. His research interest includes pervasive computing, and mobile system and applications.

Neeraj Suri received his Ph.D. from the University of Massachusetts at Amherst. He currently holds the TUD Chair Professorship at TU Darmstadt, Germany and is also affiliated with the Univ. of Texas-Austin and Microsoft Research. His earlier appointments include the Saab Endowed Chair Professorship, and Professor at Boston University. His research spans distributed systems, mobile computing and OS's tackling the design, analysis and assessment of trustworthy web scale services.