COSMOS: Computation Offloading as a Service for Mobile Devices

Cong Shi, Karim Habak, Pranesh Pandurangan,
Mostafa Ammar, Mayur Naik, Ellen Zegura
School of Computer Science
Georgia Institute of Technology, Atlanta, GA
{cshi7, karim.habak, ppandura, ammar, naik, ewz}@cc.gatech.edu

ABSTRACT

There is great potential for boosting the performance of mobile devices by offloading computation-intensive parts of mobile applications to the cloud. The full realization of this potential is hindered by a mismatch between how individual mobile devices demand computing resources and how cloud providers offer them: offloading requests from a mobile device usually require quick response, may be infrequent, and are subject to variable network connectivity, whereas cloud resources incur relatively long setup times, are leased for long time quanta, and are indifferent to network connectivity. In this paper, we present the design and implementation of the COSMOS system, which bridges this gap by providing computation offloading as a service to mobile devices. COSMOS efficiently manages cloud resources for offloading requests to both improve offloading performance seen by mobile devices and reduce the monetary cost per request to the provider. COSMOS also effectively allocates and schedules offloading requests to resolve the contention for cloud resources. Moreover, COSMOS makes offloading decisions in a risk-controlled manner to overcome the uncertainties caused by variable network connectivity and program execution. We have implemented COSMOS for Android and explored its design space through computation offloading experiments to Amazon EC2 across different applications and in various settings. We find that COSMOS, configured with the right design choices, has significant potential in reducing the cost of providing cloud resources to mobile devices while at the same time enabling mobile computation speedup.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

Computation Offloading; Mobile Devices; Intermittent Connectivity; Resource Management

MobiHoc'14, August 11–14, 2014, Philadelphia, PA, USA.
Copyright 2014 ACM 978-1-4503-2620-9/14/08 ...$15.00.
http://dx.doi.org/10.1145/2632951.2632958 .

1. INTRODUCTION

The idea of offloading computation from mobile devices to remote computing resources to improve performance and reduce energy consumption has been around for more than a decade [5, 6]. The usefulness of computation offloading hinges on the ability to achieve high computation speedups with small communication delays. In recent years, this idea has received more attention because of the significant rise in the sophistication of mobile applications, the availability of powerful clouds and the improved connectivity options for mobile devices. By identifying the offloadable tasks at runtime, recent work [11, 10, 15, 20] has aimed to generalize this approach to benefit more mobile applications.

Despite great potential, a key challenge in computation offloading lies in the mismatch between how individual mobile devices demand and access computing resources and how cloud providers offer them. Offloading requests from a mobile device require quick response and may not be very frequent. Therefore, the ideal computing resources suitable for computation offloading should be immediately available upon request and be quickly released after execution. In contrast, cloud computing resources have long setup time and are leased for long time quanta. For example, it takes about 27 seconds to start an Amazon EC2 VM instance. The time quantum for leasing an EC2 VM instance is one hour. If an instance is used for less than an hour, the user must still pay for one-hour usage. This mismatch can thus hamper offloading performance and/or incur high monetary cost.

Complicating this issue is the fact that mobile devices access cloud resources over wireless networks which have variable performance and/or high service cost. For example, 3G networks have relatively low bandwidth, causing long communication delays for computation offloading [11]. On the other hand, although WiFi networks have high bandwidth and are free to use in many cases, their coverage is limited, resulting in intermittent connectivity to the cloud and highly variable access quality even when connectivity exists [8, 12, 24].

In this paper we propose COSMOS, a system that bridges the above-discussed gaps by providing computation offloading as a service. The key premise of COSMOS is that an intermediate service between a commercial cloud provider and mobile devices can make the properties of underlying computing and communication resources transparent to mobile devices and can reorganize these resources in a cost-effective manner to satisfy offloading demands from mobile devices. The COSMOS system receives mobile user computation offload demands and allocates them to a shared set of compute resources that it dynamically acquires (through leases) from a commercial cloud service provider. The goal of COSMOS is to provide the benefit of computation offloading to mobile devices while at the same time minimizing the compute resource leasing cost.

Our goal in this paper is to develop a design for COSMOS, implement it and evaluate its performance. At the heart of COSMOS are two types of decisions: 1) whether a mobile device should offload a particular computation to COSMOS and 2) how COSMOS should manage the acquisition of compute resources from the commercial cloud provider.

We start by formulating an optimization problem whose solution can guide the required decision making. Because of its complexity, no efficient solution to this optimization is available. It does, however, lead us to the identification of three components of COSMOS decision making that we then explore individually. Specifically, we develop a set of novel techniques, including resource-management mechanisms that select resources suitable for computation offloading and adaptively maintain computing resources according to offloading requests, risk-control mechanisms that properly assess returns and risks in making offloading decisions, and task-allocation algorithms that properly allocate offloading tasks to the cloud resources with limited control overhead.

We have implemented COSMOS for Android and evaluated the system for offloading from a set of smartphones/tablets to Amazon EC2, across different applications in various types of mobile environments. We evaluate the performance of the system in several realistic mobile environments. To further explore the design space of COSMOS, we also conduct extensive trace-based simulation. In all these experiments, COSMOS achieves good offloading performance and significantly reduces the monetary cost compared with previous offloading systems like CloneCloud [10]. We find that COSMOS, configured with the right design choices, has significant potential in reducing the cost of providing cloud resources to mobile devices while at the same time enabling mobile computation speedup.

The rest of the paper is organized as follows. Section 2 presents some background material and presents the problem statement and optimization formulation. Section 3 presents an overview of COSMOS's architecture incorporating the insight gained from the optimization formulation regarding the system decomposition. The design details are presented in Section 4. COSMOS is evaluated in Section 5 and 6. We summarize related work in Section 7. Section 8 concludes the paper and discusses future work.

2. BACKGROUND AND PROBLEM STATEMENT

2.1 Background

2.1.1 Computation Offloading

A basic computation-offloading system [10, 11] is composed of a *client component* running on the mobile device and a *server component* running in the cloud. The client component has three major functions. First, it monitors and predicts the network performance of the mobile device. Second, it tracks and predicts the execution requirements of mobile applications in terms of input/output data requirements and execution time on both the mobile device and the cloud. Third, using this information the client component chooses some portions of the computation to execute in the cloud so that the total execution time is minimized. The server component executes these offloaded portions immediately after receiving them and returns the results back to the client component so that the application can be resumed on the mobile device.

Computation offloading trades off communication cost for computation gain. Previous systems [10, 11] usually assume stable network connectivity and adequate cloud computation resources. However, in mobile environments a mobile device may experience varying or even intermittent connectivity, while cloud resources may be temporarily unavailable or occupied. Thus, the communication cost may be higher, while the computation gain will be lower. Moreover, the network and execution prediction may be inaccurate, causing the performance of these systems to be degraded.

2.1.2 Cloud Computation Resources

Cloud computation resources are usually provided in the form of virtual machine (VM) instances. To use a VM instance, a user installs an OS on the VM and starts it up, both incurring delay. VM instances are leased based on a time quanta. e.g., Amazon EC2 uses a one hour lease granularity. If a VM instance is used for less than the time quanta, the user must still pay for usage. A cloud provider typically provides various types of VM instances with different properties and prices. Table 1 lists some properties and prices for three types of Amazon EC2 VM instances: *Standard On-Demand Small* instance (m1.small), *Standard On-Demand Medium* instance (m1.medium) and *High-CPU On-Demand Medium* instance (c1.medium). For some pricing models (e.g., *EC2 spot*), the leasing price may change over time.

Table 1: The characteristics of EC2 on-demand instances. The setup time is measured by starting and stopping each type of instances for 10 times. The average value and standard deviation are reported.

Instance	Cores	CPU(GHz)	Setup(second)	Price($/hr)
m1.small	1	1.7	26.5(5.5)	0.06
m1.medium	1	2.0	26.6(3.7)	0.12
c1.medium	2	2.5	26.7(8.4)	0.145

Note that the server component of offloaded mobile computation needs to run on a VM instance. This server component needs to be launched at the time the offloading request is made and terminated when the required computation is complete. The lifetime of the server component is typically much less than the lease quantum used by the cloud service provider. An important question we consider in our system design is how to ensure there is enough VM capacity available to handle the mobile computation load without needing to always launch VM instances on-demand and incur long setup time.

2.2 Problem Statement

The basic idea of COSMOS is to achieve good offloading performance at low monetary cost by sharing cloud resources among mobile devices. Specifically, in this paper our goal is to minimize the usage cost of cloud resources under the constraint that the speedup of using COSMOS against local execution is larger than $1-\delta$ of the maximal speedup that it can achieve using the same cloud service, where $\delta \in (0, 1)$. The extension of COSMOS to support other optimization goals will be discussed in Section 8. A related but independent problem is how COSMOS charges mobile devices for computation offloading, which will also be discussed in Section 8.

Let's assume that the cloud can simultaneously run N VM instances. Let's use $\langle M_i, T_i \rangle$ to denote the usage of the i^{th} VM instance, where M_i is its type (see Table 1 for examples), and $T_i = \{\langle t_{ij}, t'_{ij} \rangle | \forall j, t_{ij} < t'_{ij}, t'_{ij} < t_{i(j+1)}\}$ represents all the leasing periods. We use t_{ij} and t'_{ij} to represent the start time and end time of the j^{th} leasing period, respectively. Let $\psi(M, T)$ be the leasing cost of $\langle M, T \rangle$, which is computed by multiplying the price with the total leasing time quanta.

Let's assume that there are K computational tasks generated by mobile users in the COSMOS system during the period of time of interest. Let $O_k = \langle t_k, c_k, d_{Ik}, d_{Ok} \rangle$ denote the k^{th} computational task, where t_k ($\forall k, t_{k-1} \leq t_k$) is the time when the task was initiated by the mobile user, c_k is the number of CPU cycles it requires, d_{Ik} is the size of its input data, and d_{Ok} is the size of its output data. Let $I(i,k)$ and $I_l(k)$ be indicator functions. If O_k is offloaded to the i^{th} VM instance, $I(i,k) = 1$. Otherwise, $I(i,k) = 0$. Similarly, if it is locally executed, $I_l(k) = 1$. Otherwise, $I_l(k) = 0$. Since O_k should be executed exactly once, $I_l(k) + \sum_i I(i,k) = 1$. Let $L(O_k)$ be the local execution time of O_k which can be estimated based on c_k and the CPU frequency of the mobile device [20]. Let $R_i(O_k)$ be the response time of offloading O_k to the i^{th} VM instance, i.e., the time elapsed from t_k to the time that the output is returned to the mobile device. It depends on the network delays of sending input data and receiving output data, the execution time on the VM instance and the waiting time. Its formula will be shown in Section 4.2. The key symbols are summarized in Table 2.

Table 2: Key symbols used in COSMOS problem formulation

M_i	the type of the i^{th} VM instance
T_i	the leasing periods of the i^{th} VM instance
$\psi(M,T)$	the pricing function for leasing a VM instance
O_k	the k^{th} computational task
$I(i,k)$	the indicator function for offloading task O_k to VM i
$I_l(k)$	the indicator function for executing task O_k locally
$L(O_k)$	the local execution time of the task O_k
$R_i(O_k)$	the response time of offloading the task O_k to VM i

The maximal speedup of using COSMOS against local execution can be obtained by solving the following optimization problem:

$$\begin{aligned} \text{Max} \quad & \sum_{k=1}^{K} \frac{L(O_k)}{\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}} \qquad (1) \\ \text{s.t.} \quad & I_l(k) + \sum_{i=1}^{N} I(i,k) = 1 \end{aligned}$$

Since for each k there is exactly one of the functions $I_l(k)$ and $I(i,k)$ that equals to 1, $\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}$ equals to the corresponding $L(O_k)$ or $R_i(O_k)$, representing the response time of task O_k in COSMOS. Thus, Eqn 1 is to maximize the speedup of all tasks. Let's use S_k^* to denote the corresponding maximal speedup achieved by task O_k. We have $\forall k, S_k^* \geq 1$. The goal of COSMOS can be formally formulated as:

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{N} \psi(M_i, T_i) \qquad (2) \\ \text{s.t.} \quad & I_l(k) + \sum_{i=1}^{N} I(i,k) = 1 \\ & \frac{L(O_k)}{\min\{\frac{L(O_k)}{I_l(k)}, \{\frac{R_i(O_k)}{I(i,k)}|\forall i\}\}} \geq (1-\delta) S_k^* \end{aligned}$$

where $\delta \in (0,1)$ can be arbitrarily chosen by the system. When $\delta \to 1$, all tasks will be executed locally while the total cost will approach 0. When $\delta \to 0$, the speedups approach to the optimal values while the total cost will be high.

COSMOS is to find the values of $\langle M_i, T_i \rangle$, $I_l(k)$, and $I(i,k)$ that optimize Eqn 2. This is a challenging problem especially because we have no information regarding future computation tasks. Our approach to solve this problem is to break it down into three subproblems and address each of them separately:

- **Cloud resource management**: This is the problem of determining the number and type of VM instances to lease over time, i.e., $\langle M_i, T_i \rangle$. It has two major goals. First, there should always be enough VM instances to ensure high offloading speedup. Second, the cost of leasing VM instances should be minimized.
- **Offloading decision**: This is the problem of deciding if a mobile device offloads a computation task, i.e., deciding whether to set $I_l(k)$ to 0 or 1. The challenge comes from the uncertainties of network connectivity, program execution, and resource contention. A wrong offloading decision will both waste cloud resources and result in lower speedup. It is very important to properly handle the uncertainties.
- **Task allocation**: This is the problem of choosing a VM instance among all available instances if we decide to offload a computation task, i.e., deciding whether to set $I(i,k)$ to 0 or 1. The decision is made independently by each mobile device. In addition, the decisions for previous computation tasks will impact the decision of the current one because of resource sharing. Therefore, task allocation should be designed as a distributed mechanism.

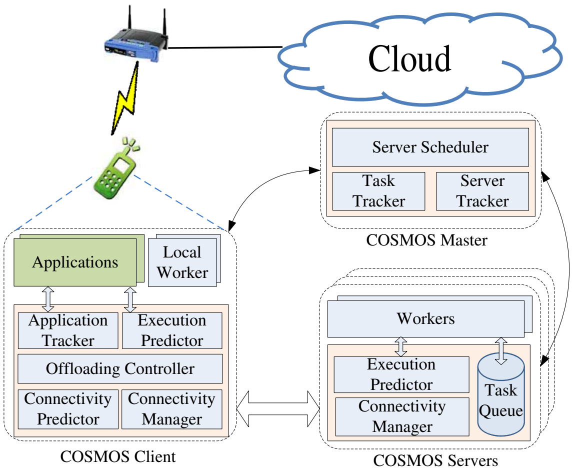


Figure 1: The architecture of the COSMOS system.

In the following sections, we will present the design of COSMOS and its mechanisms to solve these problems.

3. COSMOS SYSTEM

Figure 1 provides a high-level overview of the COSMOS system. It consists of three components: a COSMOS Master running on a VM instance that manages cloud resources and exchanges information with mobile devices; a set of active COSMOS Servers each of which runs on a VM instance and executes offloaded tasks; and a COSMOS Client on each mobile device that monitors application execution and network connectivity and makes offloading decisions.

The COSMOS Master is the central component for cloud resource management. It periodically collects information of computation tasks from COSMOS Clients through *task tracker* and the workloads of COSMOS Servers through *server tracker*. Using this information, its *server scheduler* decides the number and type of active COSMOS Servers over time. Note that when a COSMOS Server is turned on/off, its corresponding VM instance is also turned on/off. The algorithms will be described in Section 4.1.

An active COSMOS Server is responsible for executing offloaded computation tasks. Each COSMOS Server has one *task queue* and multiple *worker*s the number of which equals to its number of CPU cores. Computation tasks are executed on a first-come-first-serve basis. The COSMOS Server also estimates and provides the workload information upon request by predicting the execution time of all tasks in the queue through the *execution predictor*. We use Mantis [22], a state-of-the-art predictor for mobile applications.

A COSMOS Client tracks all applications running on its mobile device, makes offloading decisions for them, and allocates tasks to COSMOS Servers. When a mobile application starts, an *application tracker* monitors the application execution and identifies its compute-intensive tasks in the same way as MAUI [11] and CloneCloud [10]. When it reaches the entry point of such a task, the *offloading controller* obtains the computation speedup from the execution predictor and the communication delay from the *connectivity predictor* (e.g., BreadCrumbs [25]) and decides if it should offload the task based on them. The design details of this offloading decision will be described in Section 4.2. If it decides to offload, the COSMOS Client allocates the task to an active COSMOS Server, which will be described in Section 4.3. Finally the COSMOS Client offloads the task and waits to receive the result. If the COSMOS Client cannot obtain the results before a deadline, it executes the task on a local worker.

4. DESIGN DETAILS

4.1 Cloud Resource Management

The cloud resource management has two major mechanisms: how to select the type of VM instance (i.e., M_i) and when to start and stop COSMOS Servers (i.e., T_i).

4.1.1 Resource Selection

COSMOS strives to minimize the cost per offloading request under the constraint that the offloading speedup is large enough. Recall that our goal is to achieve speedup of at least $1 - \delta$ of the maximally possible. Therefore, the resource selection algorithm selects the least cost VM instance whose CPU frequency is larger than $1 - \delta$ of the most powerful VM instance. The algorithm is shown in Algorithm 1.

Algorithm 1 Resource Selection

```
procedure RESOURCESELECTION({I},$\delta$)                ▷ I is a instance type.
    maxFreq ← 0; minCost ← $+\infty$ ;
    for I in {I} do
        if maxFreq < I.f then
            maxFreq ← I.f;
        end if
    end for
    for I in {I} do
        if I.f ≥ $(1-\delta)$maxFreq && $\frac{I.p}{I.c \times I.f}$ < minCost then
            maxCost ← $\frac{I.p}{I.c \times I.f}$;
            selected ← I;
        end if
    end forreturn selected;
end procedure
```

In Algorithm 1, c, f, and p denote the number of processor cores, the CPU frequency and the price of a VM instance, respectively. If all cores are 100% utilized during a time quanta τ, the VM instance totally executes $cf\tau$ CPU cycles. The cost of each CPU cycle is $\frac{p}{cf}$. We use this value as the cost for comparison in Algorithm 1.

4.1.2 Server Scheduling

Server scheduling is the key mechanism to balance the usage cost of VM instances and the offloading performance. Its basic operations are as follows: The COSMOS Master periodically collects the number of offloading requests and the workloads of COSMOS Servers (every 30 seconds in our implementation). When the workloads are too large, it turns on new COSMOS Servers. When a time quantum of a COSMOS Server is to expire, it turns off the COSMOS Server if the remaining COSMOS Servers are enough to handle the offloading requests. The algorithm is shown in Algorithm 2.

Algorithm 2 Server Scheduling

```
 1: procedure SERVERSCHEDULING($\lambda$,$\mu$)  ▷ $\lambda$ and $\mu$ are reported arrival
    rate and service time of offloading requests in the last round.
 2:     $\lambda_s$ ← getMaxArrivalRatePerServer($\mu$, $\frac{\mu}{1-\delta}$);
 3:     n ← $\lceil \frac{\lambda}{\lambda_s} \rceil$;
 4:     if n > activeServers.size() then
 5:         turnOnServers(n - activeServers.size());
 6:     else
 7:         for s ∈ activeServers do
 8:             if s.quantumExpiring() then
 9:                 pendingServers.add(s);
10:             end if
11:         end for
12:         turnOffServers();
13:     end if
14: end procedure
15: procedure TURNONSERVERS(n)            ▷ n is the number of server
16:     for i = 1:n do
17:         if pendingServers.isEmpty() then
18:             activeServers.add(turnOnAServer())
19:         else
20:             activeServers.add(pendingServers.remove(0));
21:         end if
22:     end for
23: end procedure
24: procedure TURNOFFSERVERS
25:     for s ∈ pendingServers do
26:         if s.quantumaExpired() then
27:             turnOff(s);
28:         end if
29:     end for
30: end procedure
```

Every round the COSMOS Master computes the current arrival rate and service time of offloaded computation tasks and uses them to estimate the minimal number of COSMOS Servers to achieve the desired offloading performance. If the number of active COSMOS Servers is smaller than this value, it turns on new COSMOS Servers. A COSMOS Server whose current time quantum expires will be turned off only if the number of remaining COSMOS Servers are larger than the minimal value for several rounds. Otherwise, its lease will be renewed for another quantum.

A key function of Algorithm 2 is how to estimate the maximal arrival rate (i.e., λ_s) that a COSMOS Server can handle, as shown in Line 2. Since a COSMOS Server serves offloading requests in a first-come-first-serve manner, it can be modeled as a G/G/c system. Based on the required speedup (i.e., $1-\delta$ of the maximal speedup), we obtain that the maximal response time of the system should be smaller than $\frac{\mu}{1-\delta}$. According to Kingman's formula [19], we can obtain the value of λ_s.

4.2 Offloading Decision

The offloading controller uses the information from the connectivity and execution predictors to estimate the potential benefits of the offloading service. Ideally, if future connectivity and execution time can be accurately predicted immediately after the mobile application starts, the offloading controller can make the global optimal offloading decision. However, such global optimum is unavailable in reality.

Instead, the offloading controller uses a greedy strategy to make the offloading decision. Every time an offloadable task is initiated, the offloading controller determines if it is beneficial to offload it. Because of the uncertainties inherent in the mobile environment, the offloading decision takes risk into consideration. In case a bad decision has been made, it will also adjust its strategy with new information available.

4.2.1 *Offloading Gain*

When an offloadable task, O_k, is initiated at time t_k, the offloading controller needs to determine if it is beneficial to offload this task to the cloud. Let us use T_{ws} to denote the time to wait for connectivity before sending the data, T_s for the time to send the data, T_c for the execution time of the task in the cloud, T_{wr} for the time to wait for connectivity before receiving the result, T_r for the time to receive the result. The local execution time is $L(O_k)$, which is estimated by the execution predictor. The response time of offloading to an active COSMOS Server, $R(O_k)$, can be expressed as:

$$R(O_k) = T_{ws} + T_s + T_c + T_{wr} + T_r \tag{3}$$

It is beneficial to offload only if the local execution time is longer than the response time of offloading. Therefore, we use their difference to represent the offloading gain:

$$G = L(O_k) - R(O_k) \tag{4}$$

Because of the uncertainties in the mobile environment, the offloading controller can only obtain a distribution for G (i.e., $E(G)$ and $\sigma^2(G)$). Simply using $E(G)$ to make the offloading decision will introduce the risk of longer execution time. We describe the risk-control mechanism in the next subsection.

4.2.2 *Risk-Controlled Offloading Decision*

Our risk-controlled offloading is based on two key ideas. First, we use risk-adjusted return [9] in making the offloading decision so that the return and risk of offloading are simultaneously considered. Specifically, $E(G)$ and $\sigma(G)$ are used as the return and risk of the offloading gain, respectively. Thus, the risk-adjusted return of offloading gain is $\frac{E(G)}{\sigma(G)}$. When its value is larger than certain threshold, the computational task will be offloaded to the cloud. Otherwise, it will be locally executed. Second, we re-evaluate the return and risk when new information is available. The algorithm is shown in Algorithm 3.

When a computation task is initiated, the offloading controller evaluate its return and risk of offloading gain. Functions getOffloadingGain(O_k) and getOffloadingRisk(O_k) return $E(G)$ and $\sigma(G)$, respectively. The detailed algorithms to compute them are described in the appendix. If the risk-adjusted return (i.e., $\frac{E(G)}{\sigma(G)}$) is larger than a threshold, the offloading controller offload the task to the cloud. In addition, it also listens to the connectivity status which has high impact on $E(G)$ and $\sigma(G)$. Once new connectivity information is updated, it re-evaluates the risk-adjusted return and adjust its decision accordingly.

4.3 Task Allocation

When a COSMOS Client is to offload a task, it must decide which COSMOS Server should execute the task. We consider three heuristic methods. The first method is that the COSMOS Master maintains a global queue to directly accept offloading requests and allocates tasks when a COSMOS server has idle cores. Although it should have high server utilization, the network connecting the COSMOS Master may become a bottleneck. The second method is that the COSMOS Client queries the workloads of a set of COSMOS servers and randomly chooses one with low workload to allocate the new task. Although tasks are directly sent to COSMOS Servers in this method, it will cause huge control traffic. In addition, it will cause extra waiting time. The third method is that the COSMOS Master provides each COSMOS Client a set of active COSMOS Servers and informs it the average workloads of all COSMOS Servers. Each COSMOS Client randomly chooses a server among them to offload the task. This method has minimal control overhead. As the resource-management mechanism ensures that the workloads of COSMOS Servers are low, it should also have good performance. Thus, COSMOS uses the third method for task allocation.

Algorithm 3 Risk Controlled Offloading

```
procedure OFFLOADING($O_k$)                ▷ $O_k$ is the computation task.
    if riskAdjustedOffloading($O_k$) then
        offloadedTask ← $O_k$;
        registerReceiver(this,CONNECTIVITY);
    end if
end procedure
procedure RISKADJUSTEDOFFLOADING($O_k$)
    gain ← getOffloadingGain($O_k$);        ▷ $E(G)$ as offloading gain
    risk ← getOffloadingRisk($O_k$);        ▷ $\sigma(G)$ as offloading risk
    if gain/risk $\geq \alpha$ then
        offload($O_k$); return true;
    else
        localExecute($O_k$)
        unregisterReceiver(this); return false;
    end if
end procedure
procedure ONRECEIVE(conn)          ▷ conn is the connectivity status.
    riskAdjustedOffloading(offloadedTask);
end procedure
procedure RECEIVERESULT($O_k$)
    offloadedTask ← NULL;
    unregisterReceiver(this);
end procedure
```

5. SYSTEM IMPLEMENTATION AND EVALUATION

In this section, we evaluate our prototype implementation of COSMOS in various mobile environments.

5.1 Implementation

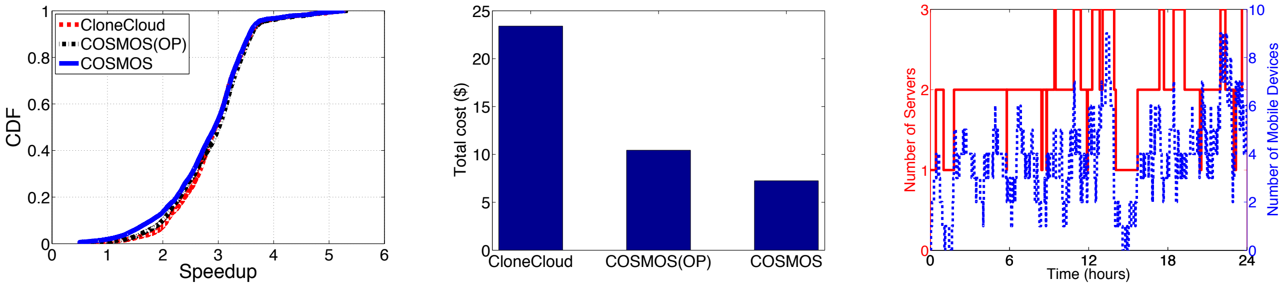
We implemented the COSMOS Server on Android x86 [2]. To run COSMOS Servers on Amazon EC2, we use an Android-x86 AMI [21] to create EC2 instances. Since Android-x86 is a 32-bit OS, three types of EC2 instances can be used for COSMOS Servers, as listed in Table 1. Based on our resource selection algorithm in Section 4.1.1, *High-CPU On-Demand Medium* instances are used to run COSMOS Servers.

The COSMOS Master runs on an EC2 instance running Ubuntu 12.04. It uses the Amazon EC2 API tools [1] to start and stop EC2 VM instances on which COSMOS Servers run.

A COSMOS Client runs on an Android device equipped with both WiFi and 3G connections. It uses the Java reflection techniques to enable the offloading of computation tasks. We modified three existing Android applications to use COSMOS, including:

FACEDETECT is a face detection application that uses APIs in the Android SDK. We collected a data set of pictures containing faces from Google Image.

VOICERECOG is an Android port of the speech recognition program PocketSphinx [18]. For simplicity of experiments, we also modified it to use audio files as input.



(a) Speedup

(b) Cost

(c) Number of active servers in COSMOS

Figure 2: The performance of COSMOS on Amazon EC2 for FACEDETECT **in one day. There are 10 Android devices which randomly become active or idle. When they are active, they randomly execute the** FACEDETECT **application.**

DROIDFISH is an Android port of the chess engine Stockfish [3] that allows users to set the strength of the AI logic.

5.2 Experimental Setup

We evaluate COSMOS in four different mobile scenarios:

- **Stable WiFi**: A student carrying a mobile device sits in his lab. WiFi is stable and fast.
- **Indoor WiFi**: A student randomly walks in a building which has good WiFi coverage.
- **Outdoor WiFi**: A student takes a campus shuttle, experiencing varying signal strength and frequent intermittent connectivity.
- **Outdoor 3G**: A student is on his commute between home and school. 3G is used for Internet access.

For each scenario, we measure the network connectivity and construct a database for the connectivity predictor.

We compare COSMOS with three baseline systems:

- **CloneCloud** is a basic offloading system in which each mobile device has a server in the cloud that is always on [10]. In addition, it assumes stable network connectivity.
- **Oracle** assumes accurate knowledge of all connectivity and execution profile information necessary to make the offloading decision. Since this is impossible to implement, we use post-analysis to obtain the results. It represents an upper-bound on offloading benefits.
- **COSMOS(OP)** is a variant of COSMOS with a simple strategy for resource management, i.e., the number of active COSMOS Servers are over-provisioned for the peak requests. We assume the number of peak requests is accurately estimated in advance.

We use two metrics to evaluate COSMOS: *speedup* and *cost*. The values of speedup are different for different mobile devices. For fair comparison, we use a Samsung Galaxy Tab running Android 2.3 to obtain the local execution time in calculating speedup. The prices for EC2 on-demand instances are used to compute the cost.

5.3 COSMOS performance with stable WiFi

This represents the best network environment for computation offloading. In this scenario, the performance of cloud resource management dominates the COSMOS performance.

In this set of experiments, we use 10 Android devices to conduct a one-day experiment for each of the systems: CloneCloud, COSMOS(OP), and COSMOS. During an experiment, each device randomly becomes active or idle from time to time. Their durations follow an exponential distribution, with average values of 0.5 hour and 1 hour, respectively. When a device is active, it randomly starts the FACEDETECT application following a Poisson distribution with arrival rate of 0.2. The same random seed is used for all three experiments.

The experiment results are reported in Figure 2. As shown in Figure 2(a), all three systems achieve similar speedups, i.e., 2.91X, 2.86X and 2.76X on average for CloneCloud, COSMOS(OP), and COSMOS, respectively. Meanwhile, the total cost of COSMOS ($7.25) is significantly lower than that of CloneCloud ($23.4) and COSMOS(OP) ($10.44), as shown in Figure 2(b).

To demonstrate how COSMOS reduces its cost, we plot the number of active devices (the dotted blue line) and that of active COSMOS Servers (the red line) in Figure 2(c). COSMOS adaptively changes the number of active COSMOS Servers according to the arrival rate of offloading requests. Therefore, COSMOS is able to reduce its cost by turning off some COSMOS Servers when the number of offloading requests is low. In contrast, COSMOS(OP) spends 44% more money to keep 3 COSMOS Servers active the whole day, whereas CloneCloud spends 223% more money than COSMOS with only 5.4% extra speedup.

5.4 COSMOS performance with variable connectivity

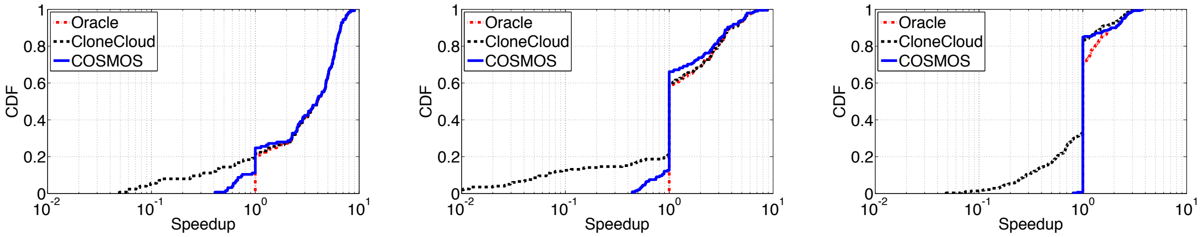
In this subsection, we evaluate how COSMOS handles variable connectivity in Indoor WiFi, Outdoor WiFi and Outdoor 3G. To eliminate the impact of cloud resource contention, we run a COSMOS Server on a machine with an 8-core 3.4GHz CPU, running VirtualBox 4.1.22, in our lab.

The dynamic mobile environment makes the comparison very hard since each invocation of an offloadable task has different Internet access quality. To achieve fair comparison, at runtime we force COSMOS to offload every offloadable task and record the information of network connectivity and application states. Then we replay these applications later for each baseline.

5.4.1 Results for different scenarios

In the first set of experiments, we evaluate the performance of COSMOS in the three different mobile scenarios using the FACEDETECT application. Figure 3 shows the speedup distribution in those experiments. When the speedup is larger than 1, the system outperforms local execution. Otherwise, the system takes longer time than local execution. In all these experiments, COSMOS performs well and achieves similar performance to Oracle. It also outperforms CloneCloud by reducing the number of bad offloading decisions.

We also find some interesting phenomena in these experiments. First, in the scenario of Indoor WiFi where mobile users have good WiFi coverage, offloading computation to the cloud benefits the mobile applications in about 80% of the cases. However, there are still about 20% in which a simple method like CloneCloud will



(a) Indoor WiFi (b) Outdoor WiFi (c) Outdoor 3G

Figure 3: A comparison of COSMOS's performance benefits using the FACEDETECT application in different mobile scenarios.

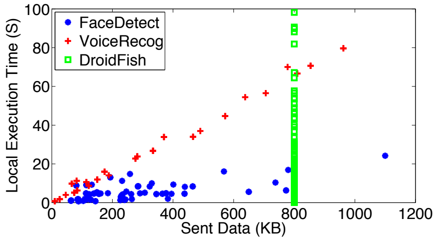
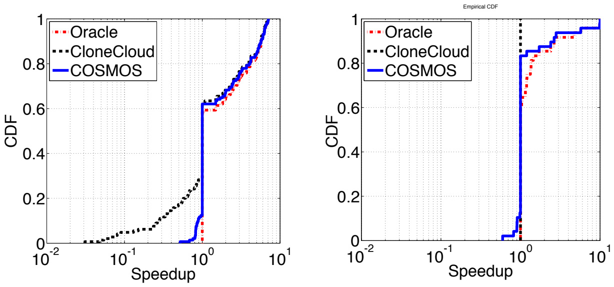


Figure 4: Local execution time vs.the size of migrated data.



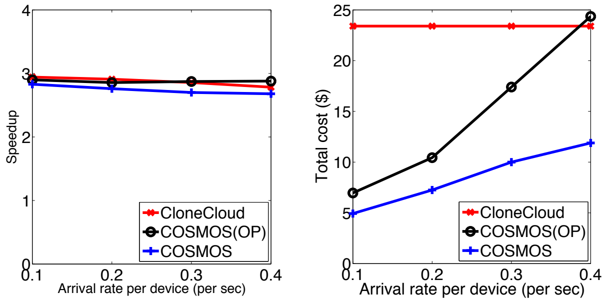
(a) VOICERECOG (b) DROIDFISH

Figure 5: COSMOS performance for different applications. All experiments are conducted in the scenario of Outdoor WiFi.

increase the execution time as much as 20 times. COSMOS reduces the portion of negative cases to about 10% with the smallest speedup at approximately 0.4. In addition, it also enables 75% of the cases to benefit from offloading. COSMOS achieves 4.1x overall speedup in this scenario. Second, in the scenario of Outdoor WiFi, CloneCloud can result in extreme speedup penalty when wrong offloading decision is made. In contrast, COSMOS still manages to limit the worst speedup results. Third, in the scenario of Outdoor 3G, ideally at most 30% of the cases benefit from offloading. This is because 3G has relatively lower bandwidth and longer delays. In this scenario, COSMOS only occasionally makes some poor decision, while CloneCloud causes more than 30% of the offloaded computation to take more time to execute.

5.4.2 Results for different applications

The gain from computation offloading is normally counterbalanced by the communication overhead. Different applications usually have different execution times and different amount of data exchanged between the mobile device and the cloud. In this subsection, we evaluate COSMOS on applications with different computation and communication properties. Figure 4 plots the local execution time of the offloadable tasks and the corresponding data to be sent to the cloud. We can see that local execution time of VOICERECOG is almost proportional to the data size, while DROIDFISH has constant data size.



(a) Speedup (b) Cost

Figure 6: Impact of the arrival rate on COSMOS performance.

To demonstrate how these application properties impact computation offloading, we compare the performance of different applications using COSMOS in the scenario of Outdoor WiFi. The results are plotted in Figure 5.

COSMOS performs well in all these experiments. For FACEDETECT and VOICERECOG, COSMOS helps more than 35% of the offloadable task invocations benefit from offloading. Meanwhile, it limits the portion of bad decision cases to be about 10% with small extra execution time. In contrast, CloneCloud makes many more bad offloading decisions and causes these invocations to last much longer.

The behavior of DROIDFISH is quite different from those of FACEDETECT and VOICERECOG. Even Oracle can only help about 15% of those invocations achieve more than 2X speedup. This is because the data uploaded to the cloud is so large that if the computation gain is small it cannot compensate for the communication overhead. We also notice that CloneCloud always chooses to execute locally because it underestimates the computation gain using previous invocations. Moreover, compared with Oracle, COSMOS does not help computations that can only achieve small performance improvement because it tries to control the risk of offloading. As a result, only a small portion of invocations have longer execution time when using COSMOS.

6. TRACE BASED SIMULATION

In this section, we use trace-based simulation to extensively evaluate the properties of COSMOS and how its components impact its performance.

6.1 Scalability

In this subsection, we analyze the impact of offload request intensity on the performance of COSMOS.

In the first set of experiments, we conduct simulation-based experiments using information logged in the experiments of Section 5.3. We vary the arrival rate of offloading requests from 0.1 to 0.4 per second and keep other settings unchanged.

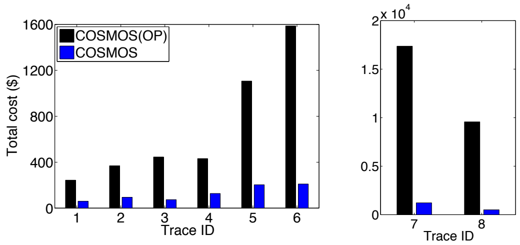


Figure 7: Offloading cost for various real-world access traces.

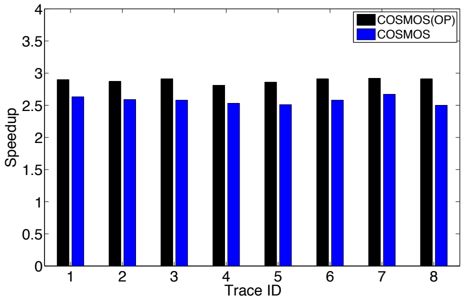


Figure 8: Offloading speedup for various real-world access traces.

Figure 6 plots the experiment results. In all experiments, COSMOS achieves the lowest cost and high speedup. We also make the following observations. First, both COSMOS and COSMOS(OP) have lower cost with lower arrival rate, while CloneCloud has constant cost. This indicates the importance of cloud resource sharing in reducing the cost. Second, when the arrival rate is very high (e.g., 0.4), COSMOS(OP) has higher speedup (i.e., 2.87X) than CloneCloud (i.e., 2.78X). Its cost is slightly higher than CloneCloud. Third, the speedup of COSMOS is similar with those of CloneCloud and COSMOS(OP) in all experiments.

Next, we evaluate COSMOS using real-world access traces [4]. The data set consists of 8 access traces each of which is composed of access requests in 2 days. We use these timestamps of access requests as the start time of mobile applications on various mobile devices. We evaluate the performance of COSMOS through simulation. The average number of requests from the same user is very low in the traces, indicating extremely high cost of CloneCloud. Therefore, we only compare COSMOS with COSMOS(OP). The costs and speedups for the FACEDETECT application on various traces are plotted in Figure 7 and 8, respectively.

COSMOS yields slightly smaller speedups but at significantly lower cost than COSMOS(OP) on all traces. Specifically, COSMOS(OP) pays 13.2 times more money than COSMOS on trace 7, while its speedup (i.e., 2.87X) is only slightly higher than that of COSMOS (i.e., 2.7X). COSMOS is able to reduce the cost by an order of magnitude while still achieving 2.7X speedup. The results of this set of experiments demonstrate that COSMOS is able to provide computation offloading with high performance at very low cost.

We also conducted experiments for our other two mobile applications, as well as a mix of all three applications. The results are similar and we omit them for brevity.

6.2 Robustness

To make the offloading decision, a COSMOS Client relies on its connectivity predictions and execution predictions to estimate how long an offloaded task will need to complete processing and return result (response time). Errors in these predictions may lead to incorrect estimation of response time and thus wrong offloading decisions and performance degradation. Here, we investigate how robust COSMOS performance is relative to such estimation errors. Because the farther the prediction is in the future the higher the absolute errors [25], we use an exponential function ($E = (-1)^r b^T$), where r is a uniform random variable and $r \in \{0, 1\}$, T is the response time. To show robustness of COSMOS and emulate different erroneous environments, we change b from 1.1 to 2. Our results are depicted in Figure 9. It shows that with the increase of the prediction error, the performance of COSMOS is degraded slightly. However, even with severe errors, COSMOS is still able to achieve high speedup for most offloading requests which highlights that the higher the offloading gains the more robustness against prediction errors.

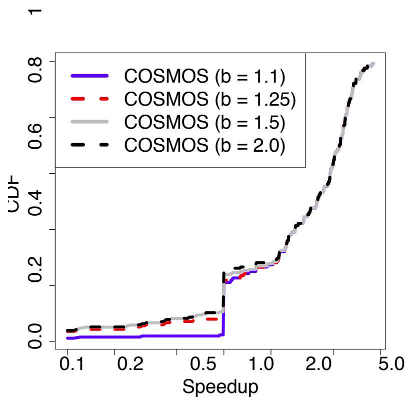


Figure 9: The impact of prediction error on COSMOS in Indoor WiFi

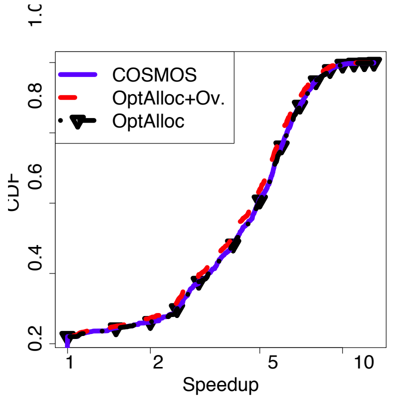


Figure 10: The performance of task allocation for Indoor WiFi

6.3 Task Allocation

To distribute the tasks across different COSMOS Servers, our task-allocation algorithm uses aggregated information about the workloads on these servers. Such aggregated information may lead to sub-optimal distribution of the tasks across the servers and affect performance. To quantify the efficiency of our task allocation algorithm, we compare it with two algorithms: 1) Optimal Allocation (OptAlloc) and 2) Optimal Allocation with Overhead (OptAlloc+Ov.). The Optimal Allocation algorithm is based on knowing the realtime COSMOS Servers queueing information and assigning a task to the server which minimizes its queuing delay. Optimal Allocation with Overhead uses OptAlloc mechanism while adding the overhead of getting the COSMOS Servers queuing information which we estimate to be one round trip time from the mobile device to COSMOS Servers. Figure 10 compares COSMOS task allocation mechanism with these two algorithms. We observe that,

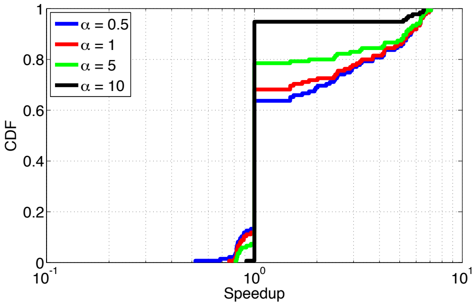


Figure 11: The tradeoff between risk and return. We use different values of α for VOICERECOG in the scenario of Outdoor WiFi.

because COSMOS ensures that the load on COSMOS Servers stays low, the Optimal Allocation mechanism did not outperform COSMOS. When the RTT overhead is accounted for, the performance degrades only slightly.

6.4 The Return-Risk Tradeoff

COSMOS enables applications to control the risk of offloading by setting the value of α. An application sensitive to extra delays can use large α value, while a small α value will result in higher expected return. To show how α impacts the return-risk tradeoff, we apply various values of α to the VOICERECOG application in the scenario of Outdoor WiFi.

Figure 11 plots the results. When the value of α increases from 0.5 to 10, the portion of invocations with speedup less than 1 decreases from about 10% to almost 0%. Meanwhile the portion of invocations that can benefit from offloading also drops from about 35% to 5%. It will be important to find a proper tradeoff between return and risk a question we relegate to future research.

7. RELATED WORK

The concept of cyber foraging [26, 13], i.e., dynamically augmenting mobile devices with resource-rich infrastructure, was proposed more than a decade ago. Since then significant work has been done to augment the capacity of resource-constrained mobile devices using computation offloading [5, 6, 16, 7]. A related technique proposes the use of cloudlets which provide software instantiated in real-time on nearby computing resources [27]. Recognizing the importance of quick responses, Ha et al. [17] propose various virtual machine techniques to enable fast provisioning of a cloudlet.

Closer to our work, MAUI [11] enables mobile applications to reduce the energy consumption through automated offloading. Similarly, CloneCloud [10] can minimize either energy consumption or execution time of mobile applications by automatically identifying compute-intensive parts of those applications. ThinkAir [20] enables the offloading of parallel tasks with server-side support. These systems focus on how to enable computation offloading for mobile devices. Based on these offloading techniques, COSMOS takes the next important step further to bridge the gap between offloading demands and the availability of cloud computing resources. In addition, COSMOS simultaneously considers performance and monetary cost in offloading. Moreover, they assume a stable network environment, whereas COSMOS also handles the more challenging mobile environment where connectivity is highly variable.

The challenges of computation offloading with variable connectivity have been identified in [28]. A system, Serendipity [29], was designed for computation offloading among intermittently connected mobile devices. In contrast, COSMOS proposes techniques to handle the variable connectivity for offloading to a cloud. A detailed survey of cyber foraging can be found in [13].

Our work is also related to studies on cloud resource management. This problem is intensively studied in the context of power saving in data centers [23, 14, 30]. For example, Lu et al. [23] uses reactive approaches to manage the number of active servers based on current request rate. Gandhi et al. [14] investigate policies for dynamic resource management when the servers have large setup time. COSMOS is different from them in three major aspects. First, they minimize the cost of power consumption, whereas COSMOS reduces the cost of leasing cloud resources. Second, in COSMOS computation tasks may be offloaded to the cloud or be executed on local devices, while in data centers services are always provided by servers. Third, COSMOS also needs to handle variable network connectivity of mobile devices, which is unnecessary for data centers.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed COSMOS, a system that provides computation offloading as a service to resolve the mismatch between how individual mobile devices demand computing resources and how cloud providers offer them. COSMOS solves two key challenges to achieve this goal, including how to manage and share the cloud resources and how to handle the variable connectivity in making offloading decision. We have implemented COSMOS and conducted an extensive evaluation. The experimental results show that COSMOS enables effective computation offloading at low cost.

There are some future directions to extend COSMOS. We will explore how to extend COSMOS to provide the computation offloading services in a manner that optimizes the energy consumption of mobile devices. It requires two major changes. First, the offloading controller should make the offloading decision based on energy consumption. It should delay computation offloading until the network connectivity is good. Second, the cloud resources can be used in a more efficient way. Instead of immediately executing each offloaded task, COSMOS should wait until enough tasks are aggregated. In addition, we will investigate the proper pricing model for COSMOS. There are several possible methods. Users pay monthly service fees and can use COSMOS as frequently as they want. Alternately, an offloaded task could be charged according to its execution time. It's even possible for mobile devices to bid for computation-offloading. When the number of offloading requests is small, COSMOS could charge a lower price to attract more requests and thereby avoid wasting cloud resources.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback. This work was supported in part by the US National Science Foundation through grant CNS 1161879.

9. REFERENCES

[1] Amazon ec2 api tools.
`http://aws.amazon.com/developertools/351.`

[2] Android-x86. `http://www.android-x86.org/.`

[3] Droidfish. `http://web.comhem.se/petero2home/droidfish.`

[4] National laboratory for applied network research. anonymized access logs.
`ftp://ftp.ircache.net/Traces/`, 2007.

[5] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *ACM SIGOPS European workshop*, 2002.

[6] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *ACM MobiSys*, 2007.

[7] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *ACM Mobisys*, pages 273–286, 2003.

[8] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *ACM Mobisys*, 2010.

[9] J. Berk and P. DeMarzo. *Corporate finance*. Addison-Wesley, 2007.

[10] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys*, pages 301–314, 2011.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *ACM MobiSys*, 2010.

[12] P. Deshpande, X. Hou, and S. R. Das. Performance comparison of 3g and metro-scale wifi for vehicular network access. In *ACM IMC*, 2010.

[13] J. Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012.

[14] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, 2012.

[15] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: code offload by migrating execution transparently. In *USENIX OSDI*, pages 93–106, 2012.

[16] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *IEEE PERCOM*, 2003.

[17] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *ACM MobiSys*, 2013.

[18] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *IEEE ICASSP*, volume 1, pages I–I, 2006.

[19] J. Kingman. The single server queue in heavy traffic. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 57, pages 902–904. Cambridge Univ Press, 1961.

[20] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *IEEE Infocom*, 2012.

[21] S. Kosta, V. Perta, J. Stefa, P. Hui, and A. Mei. Clone2Clone (C2C): Peer to Peer Networking of Smartphones on the Cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, jun 2013.

[22] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *USENIX ATC*, 2013.

[23] Y.-H. Lu, E.-Y. Chung, T. Šimunić, L. Benini, and G. De Micheli. Quantitative comparison of power management algorithms. In *DATE*, pages 20–26, 2000.

[24] R. Mahajan, J. Zahorjan, and B. Zill. Understanding wifi-based connectivity from moving vehicles. In *ACM IMC*, 2007.

[25] A. J. Nicholson and B. D. Noble. Breadcrumbs: forecasting mobile connectivity. In *ACM MobiCom*, pages 46–57, 2008.

[26] M. Satyanarayanan. Pervasive computing: Vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 2009.

[28] C. Shi, M. H. Ammar, E. W. Zegura, and M. Naik. Computing in cirrus clouds: the challenge of intermittent connectivity. In *ACM MCC*, pages 23–28, 2012.

[29] C. Shi, V. Lakafosis, M. Ammar, and E. Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *ACM MobiHoc*, 2012.

[30] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, pages 1:1–1:39, 2008.

APPENDIX

In this appendix we describe how to compute the offloading gain G of a task at time t_k. Depending on the connectivity at time t_k, T_{ws} and T_s have different distributions. In the case that the mobile device connects to the cloud at time t_k, $T_{ws} = 0$; $T_s = \frac{d_s}{b_u(t_k)}$, where d_s is the data size, and $b_u(t_k)$ is the upload bandwidth at time t_k. d_s is available at time t_k, while $b_u(t_k)$ can be estimated using the current signal strength. Otherwise, $T_{ws} = R_{D,t_k}$. $T_s = \frac{d_s}{b_u^*}$, where b_u^* is the overall upload bandwidth of the entire trace.

The value of T_{wr} depends on whether the mobile device still connects to the cloud when the cloud finishes execution at time $t + T_{ws} + T_s + T_c$. If connected, $T_{wr} = 0$. Otherwise,

$$T_{wr} = \begin{cases} D - R_{C,t_k} + T_s + T_c, & \text{if connected at } t_k \\ D - C + T_s + T_c, & \text{otherwise} \end{cases} \tag{5}$$

where C and D are contact duration and inter-contact duration, respectively.

T_r also depends on the connectivity at time $t' = t_k + T_{ws} + T_s + T_c$. If connected, $T_r = \frac{d_r}{b_d(t')}$, where d_r is the result size, and $b_d(t')$ is the download bandwidth. Otherwise $T_r = \frac{d_r}{b_d^*}$, where b_d^* is the average download bandwidth.

According to the above analysis, T_{wr} is directly related to T_s and T_c. T_r is indirectly related to T_s and T_c as $T_s + T_c$ may impact the distribution of signal strength which impacts T_r. However, this correlation is small and, thus, be ignored in the implementation for simplicity. Other variables are independent of each other. Therefore, the variance of offloading gain can be computed using

$$\begin{aligned} \sigma^2(G) &= \sigma^2(T_{ws}) + \sigma^2(T_s) + \sigma^2(T_c) + \sigma^2(T_{wr}) \\ &\quad + \sigma^2(T_r) + \sigma^2(T_l) + 2\sigma(T_s + T_c, T_{wr}) \end{aligned} \tag{6}$$