Collaborative Offloading for Distributed Mobile-Cloud Apps

Hillol Debnath*‡, Giacomo Gezzi†‡, Antonio Corradi†,
Narain Gehani*, Xiaoning Ding*, Reza Curtmola*, and Cristian Borcea*
*Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
†Department of Computer Science and Engineering, University of Bologna, 40126 Bologna BO, Italy
Email: *{hd43, gehani, xiaoning.ding, reza.curtmola, borcea}@njit.edu,
†{giacomo.gezzi@studio.unibo.it,antonio.corradi@unibo.it}

***Abstract*—Computation offloading has been widely used to improve the energy consumption and completion time for standalone apps in mobile-cloud platforms. However, existing approaches have not been designed for distributed mobile-cloud apps and, thus, they are unable to provide effective solutions for such apps that have job and device dependencies, specific to their distributed nature. This paper presents CASINO, a dynamic and collaborative computation offloading framework which employs distributed profiling, decision making, and job execution to achieve an optimized completion time of the distributed computation. CASINO's main component is its job scheduler that works in real-time and considers the global resource conditions and job/device dependencies in order to generate an optimized job schedule for a distributed app. We validated this scheduler by using simulated albeit realistic data. We also built a prototype of CASINO and evaluated it using a proof-of-concept distributed app. The results show that CASINO can significantly improve the computation latency when compared to solutions that execute all offloadable jobs on mobile devices or in the cloud.**

***Index Terms*—mobile-cloud, computation offloading, mobile distributed systems**

I. Introduction

Mobile-cloud computing is expanding from supporting standalone mobile apps to supporting distributed apps [1], [2]. Given the potential scale of these apps, it is important to make them as efficient as possible. In this paper, our goal is to optimize the completion times of these apps. One way to achieve this goal is to create an effective and usable offloading system that takes into account two main characteristics of distributed mobile-cloud apps: (i) run over heterogeneous devices (in terms of computation, network, energy, sensors, and context), and (ii) have job and device dependencies. This framework must be transparent to users and involve minimal actions from programmers.

Many studies [3]–[7] have provided efficient frameworks for offloading computation from mobile to cloud for standalone (i.e., non-distributed) apps. Since these frameworks were not designed to work for distributed apps running over heterogeneous devices, they cannot optimize the execution time of such apps. In order to complete the whole distributed app as fast as possible, it is essential to have a job scheduler that considers job/device dependencies and dynamically utilizes all available computing resources to parallelize the execution of the jobs to the extent possible.

We have designed and implemented CASINO, a dynamic and collaborative offloading framework for distributed apps running on mobile-cloud computing platforms such as Avatar [1]. CASINO has a job scheduler that works in real-time and considers the global resource conditions and job/device dependencies in order to generate an optimized job schedule for a distributed app. This framework provides a simple API set that can be used by programmers to partition their apps statically and dynamically. For static partitioning, programmers can annotate their jobs with "local" or "remote" to request the framework to execute them on the mobile or in the cloud, respectively. For example, the programmers know that a job needs access to a sensor available only on the mobile device or that the job accesses private data that must not be transferred to the cloud. On the other hand, some jobs may be computationally-intensive while requiring little data; as such, they are good candidates to be executed in the cloud. For jobs that have no clear requirements to run on the mobile or in the cloud, the programmers employ dynamic offloading by marking these jobs as "offloadable".

The framework profiles resource information (e.g., CPU, battery level, bandwidth, data communication cost, etc.) from participating devices in order to make its scheduling decisions. It is worth noting that the cost is prohibitive to find an optimal schedule that minimizes the total completion time of the whole computation because this is an NP-hard problem [8]. Therefore, we have designed a heuristic solution, which can generate a good schedule in polynomial running time. Our scheduler reduces the exponential search space (e.g., all device-job combinations) to a polynomial range through a combination of topological sorting and greedy scheduling. An ordered sequence of jobs is generated by topological sorting based on job dependencies. The scheduler will then try only the (device, job) combinations from this sequence. The greedy method is applied for each job to select the best device available to minimize its completion time. Our scheduler ensures that all *offloadable* jobs execute on mobile devices or in the cloud in such a way as to reduce their computation and communication time. Hence, the total completion time of the distributed app is reduced.

‡ Authors have equal contributions.

We have developed a prototype of CASINO in Android and evaluated it in two phases. First, the job scheduler is validated using a simulated data set which is modeled using realistic device and network conditions. The validation shows that the scheduling algorithm results in better completion time compared to scenarios where: 1) all jobs are executed in the cloud, or 2) all jobs are executed on mobile devices. The second part of the evaluation is done using micro-benchmarks to assess whether CASINO's execution engine carries out the offloading efficiently. The results show that CASINO is capable of executing computation offloading with very low energy usage and overhead.

To summarize, this paper has three major contributions: (1) To the best of our knowledge, we designed and implemented the first framework for collaborative offloading of distributed mobile-cloud apps; (2) We created an effective scheduling algorithm for offloading jobs of a distributed mobile-cloud app, which takes into account job/device dependencies and device heterogeneity; (3) We demonstrated the effectiveness and efficiency of our approach through experiments using an Android prototype and realistic simulations.

The rest of the paper is organized as follows. Section II discusses related work. Section III presents the overall design of CASINO, and a brief overview of the Avatar system for which it is implemented. The job scheduling problem for mobile-cloud app offloading and our scheduling mechanism/algorithm for this problem are described in Sections IV. The prototype implementation is presented in Section V. Section VI shows the evaluation results. The paper concludes in Section VII.

II. Related Work

Computation offloading has been studied in many works to improve the execution speed for face recognition [3], [9], image search [4], gesture and object recognition [10], [11], video encoding and transcoding [12], [13], and speech recognition [14]. A native code offloader is presented in [12], [15] to offload code written in C++. "Ready, Set, Go" [16] presents a framework that bundles offloadable tasks from multiple application to save network/energy usage. Eom et al [17] describe a framework that employs a machine learning mechanism to decide how the code should be partitioned at run-time. Bhattacharya et al. summarize additional offloading works in [7]. All these works differ from CASINO in one important aspect: CASINO is designed to work with *distributed* mobile-cloud apps, whereas these other works are designed for each standalone app running on a single mobile-cloud pair.

A few works have also considered a multi-device setup. The work in [18] proposes a model that takes into account the risk factors and reliability of offloading in a multi-node setup. The works in [19], [20] present a solution to minimize energy consumption, when multiple devices are trying to offload using the same cellular access point. The research in [20], [21] employs game-theoretic approaches to minimize the energy usage for a group of users. These works are different from CASINO in that they do not consider a collaborative computation scenario, where all the jobs executed by different users are parts of a larger distributed computation.

III. Offloading Distributed Mobile-Cloud Apps

A. Avatar Platform for Distributed Mobile-Cloud Apps

Although the concept of CASINO is generic and can be implemented on any mobile-cloud distributed platform, we have implemented it to work on top of the Avatar [1] platform and the Moitree [22] middleware of the Avatar platform. Avatar is designed to leverage cloud resources to support fast, scalable, and energy efficient distributed computing over mobile devices. Each user has a virtual machine (called *avatar*) in the cloud working as the surrogate of her mobile device, which assists the execution and communication of the user's mobile apps. An avatar runs the same operating system as the mobile device. Therefore, the application code can be executed unmodified on both the mobile and the avatar.

Programmers can use the Moitree [22] API to create groups to organize participants in order to execute a distributed computation. Moitree facilitates the distributed execution environment by taking care of group communication (e.g., distributing data, fetching results) and managing the pairing between each mobile device and its associated avatar. Moitree also provides communication support for the offloading framework. Computation and its accompanying data is offloaded to or from the avatar via Moitree's communication API.

B. CASINO Overview

In a mobile-cloud platform such as Avatar, a distributed app is executed on a combination of mobile devices and cloud entities (avatars) of a group of users. The distributed app comprises of many smaller jobs. The overall completion time of the distributed computation thus depends on how fast the smaller jobs are executed by utilizing both local and cloud resources. In other words, a well coordinated usage of computation offloading can significantly improve the total completion time of the distributed computation.

CASINO optimizes the total completion time of a distributed app by scheduling, offloading, and executing its jobs in an efficient manner. CASINO provides a simple programming framework which can be used by programmers to partition their apps both statically and dynamically. Although static partitioning is generally less effective, it can be helpful in some situations. For example, user-interaction tasks should be statically partitioned to the mobile devices. On the other hand, programmers can annotate parts of their code as "offloadable" to utilize dynamic partitioning. Due to device/job dependencies as well as communication latency, executing all offloadable jobs to the cloud does not work well in many situations. Therefore, CASINO provides a dynamic scheduling that takes into account all these constraints.

CASINO profiles user devices to collect networking, CPU, and battery status. This profiling information is sent to a cloud service named job scheduler (shown in Figure 1). The job scheduler does not handle jobs directly, which are the code components and data residing on user platforms (mobile

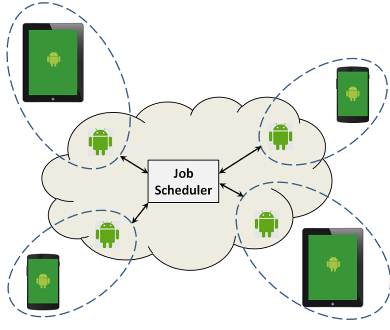


Fig. 1: CASINO's Job Scheduler Serves a Distributed App Running on Several Mobile-Avatar Pairs

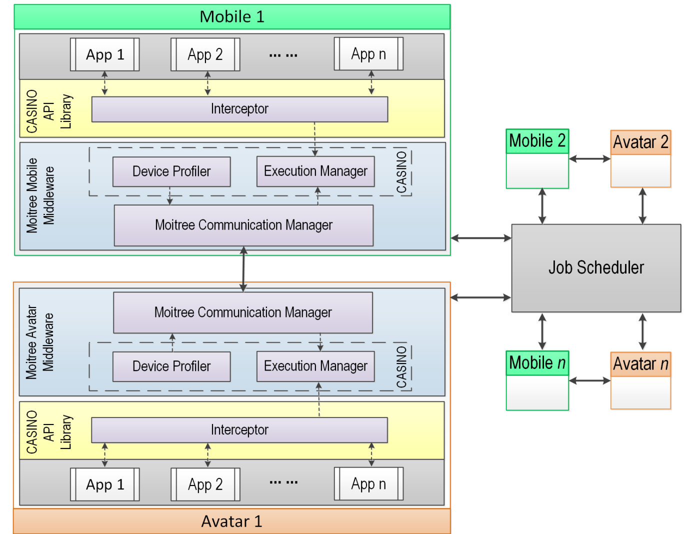


Fig. 2: CASINO Architecture

devices or avatars). It only handles the representations of jobs, which are just objects organized locally by the job scheduler. The objects hold the references to the actual jobs and meta-data information of the jobs.

When a distributed app starts execution, CASINO enqueues the representations of all the *offloadable* jobs in a job queue. The scheduler accounts for device dependency of all jobs, and then it resolves the dependencies between jobs. Next, it calculates an ordered sequence of jobs such that each job starts its execution only after its predecessor job has finished. The scheduler then estimates the best execution time of all jobs considering both device and job dependencies. The execution time estimation also includes any data communication delay associated with the job execution. For example, jobs J_1, J_2 are estimated to run fastest on $mobile_1$ and $avatar_2$ respectively. J_2 depends on the output of J_1 and there is a communication cost (i.e., delay) to deliver the output data from $mobile_1$ to $avatar_2$. The job scheduler will schedule jobs considering both these computation and communication costs. Hence, CASINO provides dynamic partitioning of a distributed app and reduces the app completion time.

C. CASINO System Architecture and Key Components

CASINO has four main components: API Library, Device Profiler, Execution Manager, and Job Scheduler. The architecture of CASINO and its components are shown in Figure 2.

1) API Library: CASINO has a simple API set for partitioning code statically and dynamically. Programmers can annotate code components (e.g., classes or functions) using *@Local* or *@Remote* tags to partition code statically. Dynamic partitioning is achieved by annotating code components with *@Offloadable* tags. These tags instruct CASINO on how to schedule and execute the code components. Specifically, *@Local* tags instruct CASINO to schedule the jobs on their host mobile devices; *@Remote* tags instruct CASINO to schedule the jobs on avatars; and *@Offloadable* tags give CASINO the freedom to schedule the jobs on their host mobile devices, remote mobile devices, or avatars. It can be noted that any code component without annotation is executed locally. In the API library, a $CodeInterceptor$ is used to catch the start of the annotated parts of the code, generate jobs, and forward a list of *offloadable* code components (i.e., jobs) to the job scheduler.

2) Device Profilers: CASINO runs a profiler on each mobile device or avatar to monitor and collect network bandwidth and latency, battery level, CPU capability and usage, and memory capacities in these devices/avatars. The profiler sends the collected information to the job scheduler.

3) Job Scheduler: The core of CASINO framework is its job scheduler, which is designed as a cloud service. Mobile devices and avatars communicate with the job scheduler to periodically submit jobs (i.e., job representations) as well as profiling information. The scheduler organizes the jobs it receives using a job queue, and runs a greedy algorithm to decide where each job should be executed. It forwards the jobs assigned to each device/avatar to the corresponding execution managers in this device/avatar for execution. The detailed scheduling algorithm is described in Section IV.

4) Execution Manager: CASINO uses an execution manager on each mobile device/avatar to receive the jobs assigned by the job scheduler, manage the data required by the jobs, and control their executions.

IV. Job Scheduling

A. Scheduling Problem Formulation

In the scheduling problem, a set of jobs $J = \{J_1, J_2, ..., J_n\}$ must be assigned to a number of devices $M = \{D_1, D_2, ..., D_m, Av_1, Av_2, ..., Av_m\}$, which consist of a mobile device D and an avatar Av for each user. An algorithm is needed to produce a schedule, s[][] = $\{s_{ij} | i \epsilon M, j \epsilon J\}$, where $s_{ij} = 1$ indicates job j is scheduled on device i. A good schedule is one that can reduce the overall completion time of the jobs without violating any job constraints.

In CASINO, the constraints are job dependencies to devices or to each other. Device dependencies define where the jobs must run. They are incurred when a job needs certain resources

on particular devices, e.g., the accesses to files or sensors. Job dependencies define when the jobs can start execution. They are incurred when a job needs the output of other jobs. The dependencies of a job must be satisfied before the job can start execution.

As an example, a distributed app allows its users to search the faces of their friends in a publicly available data set. In the first phase, faces are detected in the photos of the data set. Since there are no privacy issues with the data set, the jobs can run on any mobile device or avatar. In the second phase, face recognition is done on the detected faces and the personal photos of their friends. These jobs depend on the output of the face detection jobs (i.e., job dependencies). At the same time, since personal photos must be used and some of the photos may only be accessible on the corresponding mobile devices due to privacy concerns, the related jobs must be executed on the corresponding mobile devices (i.e., device dependencies).

Machine dependencies can be determined by analyzing the *@Local* and *@Remote* annotations used by the programmer. They are described using a matrix $R = \{R_{ij} | i\epsilon M, j\epsilon J\}$, where $R_{ij} = 1$ indicates that job j is dependent on device i. If $R_{ij} = 0$, then job j is not dependent on device i. If job j does not have dependency on any device, it can be executed on any mobile or avatar of the users participating in the distributed computation. Job dependencies can be detected using a code analysis tool [23] and are described using a matrix $E = \{e(j,k) |$ job J_k is dependent on job $J_j\}$.

In CASINO, the scheduling algorithm considers both computation cost and communication cost. The computation cost is managed using a computation cost matrix $COMP = \{COMP_{ij} | i\epsilon M, j\epsilon J\}$, where $COMP_{ij}$ indicates the cost (time) of computing job J_j on device M_i. For pairs of jobs/devices that have been profiled already, the cost is known. For pairs that have not been profiled yet, the cost can be estimated using the instruction counts of jobs and the CPU capability of the devices. In Android, instruction counts of a function can be calculated by using the Android Debug API. The system can also improve the cost estimation over time using profiling information.

The communication cost is estimated using the amount of data exchanged between jobs and the time to transfer a unit of data between the devices. The amount of data exchanged between jobs is managed by the matrix $IN = \{IN_{ij} | i\epsilon M, j\epsilon M\}$, where IN_{ij} indicates the size of input data needed by job J_j from J_i. The time to transfer a unit of data is managed using a communication overhead matrix $COMM = \{COMM_{ij} | i\epsilon M, j\epsilon M\}$. The values in the above matrices can be obtained by profiling the current execution of the app and monitoring the devices/avatars. They can also be adjusted based on profiling information collected during the past executions of the app to improve the accuracy of estimation.

B. CASINO Scheduling Algorithm

Based on the $\alpha|\beta|\gamma$ model of Lawler et al. [8], the problem of scheduling the jobs in a distributed mobile-cloud app is to assign jobs with dependencies (i.e., $|prec|$ for the β part of the model) to m uniform parallel machines (i.e., Q_m for the α part of the model) with an optimization goal of minimizing the total completion time (i.e., $\sum_{j=1}^{n} C_j$ for the γ part of the model). The problems of the $Q_m|prec|\sum_{j=1}^{n} C_j$ class are NP-hard [8]. To reduce the complexity, we have designed a greedy algorithm to find a good scheduling with low overhead. The greedy scheduling algorithm generates schedules in polynomial time. Although it cannot generate an optimal solution, it achieves near-optimal results in a realistic time frame, which is essential for a dynamic offloading scheduler. Our evaluation shows that using a greedy algorithm is a reasonable compromise between schedule optimality and execution overhead.

Algorithm 1 CASINO Job Scheduling Algorithm

```
1:  procedure SCHEDULE(J, M, E, R)
2:      J: jobs; M: devices;
3:      E: job dependencies; R: device dependencies
4:      L ← topologicalSort(E)
5:      for each job j in L do
6:          for each device d in M do
7:              estimate COMP[d][j]
8:          end for
9:      end for
10:     for each job j in L do
11:         if j.isDeviceDependent() then
12:             d ← j.deviceDependency()
13:             s[d][j] ← 1
14:             continue
15:         end if
16:         for each available device d in M do
17:             schedule.cost ← COMP[d][j] + communicationCost(d, j)
18:             schedule.device ← d
19:             minHeap ← schedule
20:         end for
21:         d ← minHeap.poll().device
22:         s[d][j] ← 1          ▷ device d is optimal to schedule j
23:     end for
24:     return s                 ▷ Return the schedule
25: end procedure
```

CASINO schedules jobs periodically in batches. In each batch, the scheduler receives a list of jobs from the job queue. The scheduling of the jobs in a batch is shown in Algorithm 1.

The algorithm first uses a topological sort (Line 4) to get an ordered list (L) of jobs based on their job dependencies. The jobs without any dependencies are organized at the beginning of the list, and the jobs depending on other jobs (i.e., predecessor jobs) are arranged after their precedent jobs. Topological sort is a standard graph algorithm when the job dependency graph is available, and is not elaborated for brevity.

Then, the algorithm estimates the computation cost for each job on every possible device, and builds the matrix $COMP$ (shown in Line 7), with $COMP[d][j]$ being the estimated

Algorithm 2 Calculation of Communication Cost for State Synchronization

```
1: procedure COMMUNICATIONCOST(d_1, j)
2:     cost ← 0
3:     L_pred ← predecessor(j)
4:     for each job k in L_pred do
5:         d_2 ← scheduled_device(j)
6:         cost ← cost + COMM[d_2][d_1] * IN[k][j]
7:     end for
8: end procedure
```

execution time of job j on device d.

After that, the scheduler iterates through all jobs in the ordered list L. For each job j, it first checks whether the job has a device dependency (Line 11). If it does, the job is scheduled on the corresponding device d. Otherwise, the algorithm tries to find an available device d, where the job can be finished fastest. This is done by calculating and comparing the execution cost of the job on every possible device, which consists of both the computation cost and the communication cost. The comparison is done using a heap in the algorithm (shown with the minHeap in Line 19). After device d has been selected, the s[d][j] entry of the schedule matrix is updated to reflect the schedule.

In the above algorithm, the communication cost incurred by scheduling a job to a device is the time spent on transferring the output data of the predecessors of the job to the device. The cost is estimated with Algorithm 2 based on the amount of data transferred (i.e., IN entries) and the communication overhead (i.e., $COMM$ entries) between the corresponding devices. The function *communicationCost* first finds the list of the predecessor jobs of job j (shown in Line 3). As shown in Line 6, it then estimates the communication costs from the devices where this jobs run to the current device of job j. The function returns the sum of all such costs.

Some devices (e.g., a few powerful avatars) may get selected repeatedly for multiple jobs. To prevent such devices from being overloaded, the algorithm maintains for each device a ratio between the CPU capacity and number of jobs already assigned to the device. If this ratio exceeds a particular threshold, the device is marked as unavailable until it finishes its jobs. In such a situation, Algorithm 1 retrieves the next best device in Line 21.

After iterating through all jobs in the ordered list L, the schedule matrix contains the updated schedule, which is returned as the output of the algorithm in Line 24. The time complexity of the scheduling algorithm is O($mn^2 logm$), where n is the number of jobs and m is the number of devices.

V. System Implementation

We have implemented a prototype of CASINO using Java and Android SDK as a part of the Moitree middleware [22]. The CASINO instance on each device/avatar uses a TCP library named Kryonet [24] to communicate with other CASINO instances and the job scheduler. We select Kryonet because it provides fast data serialization, which helps to reduce the overall latency of data communication. CASINO uses Android API to obtain resource information (e.g., CPU computing capability) in mobile devices. It also uses tools, such as Battery Historian [25], Network Monitor [26], and HPROF Analyzer [27], to profile the capacity and usage of other resources, including battery, network, and memory.

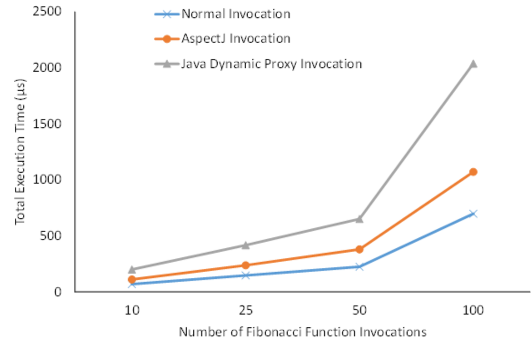


Fig. 3: Execution Time for Test Program when Invoking a Method via Normal Java Calls, Java Dynamic Proxy, and AspectJ

Since CASINO API library and middleware are different processes, we use Android's binder IPC interface to facilitate inter-process communication among them. The library uses the Java annotation processing and AspectJ [28] to intercept the creation of jobs with different annotations (i.e., *@Local*, *@Remote*, or *@Offloadable*). AspectJ uses the Aspect-Oriented Programming paradigm [29] and supports enhancing existing code with code injection during compile time. The interception can also be achieved by using Java Dynamic Proxy mechanism. However, it is inherently slower and incurs higher overhead compared to the compile-time code injection with AspectJ. We have tested the overhead with an experiment, in which a Fibonacci function is repeatedly invoked in a program to make this program computationally intensive, as the offload code is expected to be. Fig 3 shows how the execution time of the program changes when the function is invoked for different numbers of times for three different invocation methods: i) normal Java invocation, ii) invocation with Java Dynamic Proxy, and iii) invocation with AspectJ. The results clearly show that AspectJ incurs substantially lower overhead than the Dynamic Proxy method.

VI. Evaluation

We have conducted two sets of experiments. The first set of experiments validate the effectiveness of the CASINO job scheduler with simulation and a synthetic data set. We show the schedule generated by the scheduler leads to lower completion time, compared to running all the jobs on mobile devices and running all the jobs on avatars. The second set of experiments evaluate the efficiency and performance of CASINO's execution manager with a proof-of-concept application named PhotoFilter. We show that the execution manager incurs minimal overhead.

A. *Effectiveness of the Job Scheduler*

Due to the lack of real profiling data and the difficulty of collecting such data, we validate the job scheduler with a synthetic data set. The data set is generated to reflect the execution of the friend-finding face recognition app described in Section IV-A. The evaluation tests the scheduling of one batch of jobs, since the performance is the same across different batches.

The data set used in the evaluation is as follows. Eight jobs of three users are scheduled on up to 6 machines. The machines with indexes from 0 to 2 are mobile devices, and the machines with indexes from 3 to 5 are avatars.

The device dependency matrix is

$$\text{D} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each row in the above matrix represents a machine, and each column represents a job. Entry D(i,j) indicates whether job j depends on device i or not. A value of 1 indicates j is dependent on i, thus j should be scheduled on machine i. A value of 0 indicates a lack of dependency. Job J_0 is the initial job, and J_7 is the final job merging and reporting results. They must run on machine M_0.

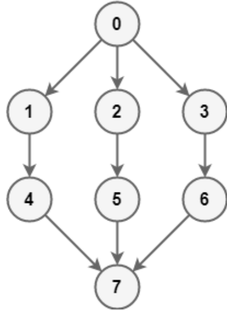


Fig. 4: The Job Dependency Graph for Validation App

The dependencies between the jobs are as shown in Figure 4, and can be described using the following job dependency matrix:

$$\text{E} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

An entry E(j,k) in the matrix tells whether job k depends on job j (value = 1) or not (value = 0). For example, jobs J_1, J_2, and J_3 depend on job J_0.

The communication cost matrix is

$$\text{COMM} = \begin{bmatrix} 0 & 250 & 300 & 30 & 32 & 30 \\ 250 & 0 & 290 & 38 & 30 & 31 \\ 300 & 290 & 0 & 42 & 33 & 38 \\ 30 & 38 & 42 & 0 & 5 & 3 \\ 32 & 30 & 33 & 5 & 0 & 7 \\ 30 & 31 & 38 & 3 & 7 & 0 \end{bmatrix}$$

$COMM$ is a 6x6 matrix, since there are 6 machines. An entry COMM(i,j) indicates the time in milliseconds for transferring 1KB of data from machine i to machine j. Mobile to mobile communication is most costly (250-300ms). The communication cost between mobile devices and the cloud is moderate (30-42ms). The cost of avatar to avatar communication is very low (within 3-7ms), since we assume avatars are physically located in the same data center. These values are modeled based on common values encountered in wireless networks and data centers. We also assume that all jobs need 1KB data from their predecessor jobs. The input/output matrix IN is not shown for brevity.

The computation cost matrix is estimated as:

$$\text{COMP} = \begin{bmatrix} 50 & 400 & 400 & 250 & 250 & 250 & 250 & 100 \\ 400 & 400 & 400 & 250 & 250 & 250 & 250 & 250 \\ 400 & 400 & 400 & 250 & 250 & 250 & 250 & 250 \\ 70 & 70 & 70 & 40 & 40 & 40 & 40 & 150 \\ 70 & 70 & 70 & 40 & 40 & 40 & 40 & 150 \\ 70 & 70 & 70 & 40 & 40 & 40 & 40 & 150 \end{bmatrix}$$

The COMP matrix has a size of 6x8. Each row represents a machine and each column represents a job. An entry COMP(i,j) refers to the estimated computation time in milliseconds of job j if it is executed on machine i. For example, job J_2 will take 400ms if it is executed on mobile device M_0, and will take 70ms if it is executed on an avatar M_3.

Based on the aforementioned data set, the job scheduler produces the output schedule shown below:

$$\text{s} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Except for job J_0 and job J_7, which must be scheduled on machine M_0 (user U_0's mobile), as the schedule indicates, job J_5 and job J_6 should be scheduled on M_1 (user U_1's mobile) and M_2 (user U_2's mobile), respectively; jobs J_1 and J_2 should be scheduled together on machine M_3 (user U_0's avatar); jobs J_3 and J_4 should be scheduled on M_4 (user U_1's avatar) and M_5 (user U_2's avatar), respectively.

Table I compares the total completion times of the jobs for three schedules: 1) the schedule generated by CASINO, 2) a schedule with all the jobs running on the mobiles, and 3) a schedule with all jobs scheduled on avatars. The total completion time is the lowest when the jobs are scheduled by the CASINO scheduler. This shows the effectiveness of

TABLE I: Comparison of Total Completion Time for Three Possible Schedules (in ms)

Scheduled by CASINO	Everything Scheduled On Mobiles	Everything Scheduled On Avatars
1614	3416	1837

the CASINO scheduler and the benefits of dynamic code offloading in a distributed mobile-cloud app.

B. *Efficiency of CASINO's Execution Manager*

We have implemented two versions of an app named PhotoFilter to verify the performance benefits of mobile-to-cloud job offloading and the efficiency of CASINO's execution manager.

The PhotoFilter app allows a user to select an image and apply different types of graphic filters such as blurring, gray-scale rendering, or inverting the colors. The filtering jobs are characterized by heavy computation. The blur filter is implemented by a Gaussian function with a user defined radius. The computation can be done locally on a mobile device or be offloaded to an avatar and executed in the cloud. When the computation is offloaded to the cloud, the photos must also be transferred to the cloud to be processed there.

CASINO provides support for both standard Android code (written in Java) and native C++ code. Thus, we implemented two different versions of the same Gaussian function, one in Java and the other in C++. Figure 5 shows the execution time of the blur filter when applied on one image using these two implementations.

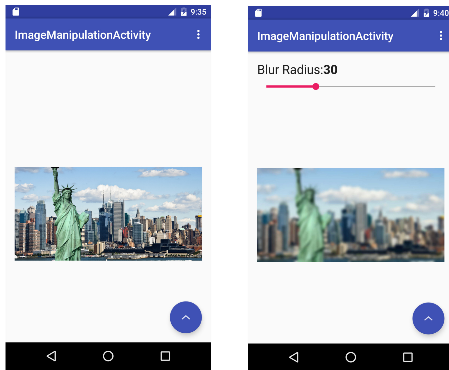
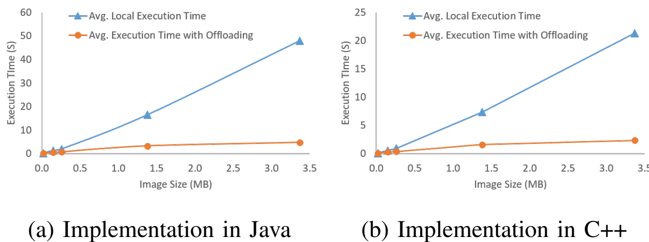


Fig. 5: Using the Blur Filter in the PhotoFilter App



(a) Implementation in Java (b) Implementation in C++

Fig. 6: Execution Time of Blur Filter with and without Offloading Support

TABLE II: Overhead Introduced by CASINO's Execution Manager

State Size (Kb)	Execution Time Including Offloading (ms)	Overhead - Interception and State Initialization (ms)	Overhead - State Sync (ms)	Overhead Percentage
237.31	3212	2.11	0.06	0.06%
61.36	664	2.75	0.07	0.40%
36.44	490	2.93	0.08	0.61%
6.70	145	2.79	0.06	1.97%

We observe that the benefits of offloading the computation to the cloud increase with image sizes. For small photos, offloading the computation to the cloud does not improve performance. The lower computation time is offset by the overhead of state synchronization. Large photos take longer time to process on mobile devices. Thus, offloading this computation can significantly lower the processing time. It can also be noted that image manipulation and signal processing have a huge number of array operations that are inherently slower with Java than with C/C++ [30]. For this reason, the C++ implementation runs faster than the Java implementation on both Android devices and avatars.

We have also measured the overhead introduced by CASINO's execution manager. The overhead is incurred mainly by 1) the interception of method invocations and state initialization (i.e., extracting and preparing the state of run-time objects that would be transferred), and 2) transfer of memory states locally between the app and the middleware. To highlight the overhead, we have deliberately increased the state sizes.

Table II shows the overhead introduced by CASINO's execution manager. For different state sizes (shown in column 1), the time needed to execute the offloaded computation varies dramatically (column 2). However, the overhead incurred by code interception and state initialization is minimal and kept almost stable (column 3). The overhead needed to transfer the states locally (column 4) is even lower than the overhead of interception and state initialization. Across all the state sizes, the overhead is less than 2% of the execution time. This indicates that the runtime overhead of CASINO is very low.

The major overhead of computation offloading is the data communication between the mobile device and the avatar. We have tested the communication overhead on a WiFi network for different data sizes. As shown in Table III, the communication cost is generally negligible, compared to the execution times for computationally expensive apps.

TABLE III: Communication Overhead of CASINO

State Size (Kb)	Execution Time Including Offloading (ms)	Communication Time (ms)
237.31	3212	13
61.36	664	4
36.44	490	2
6.70	145	0.1

VII. Conclusion

To the best of our knowledge, this paper presented the first computation offloading framework for distributed apps running on mobile-cloud platforms. This framework, CASINO, provides an API set for statically and dynamically partitioning code and marking offloadable code components. CASINO's job scheduler optimizes the app completion time and schedules all the offloadable code components with the constraints determined by job dependencies and device dependencies. The scheduler uses a greedy algorithm to generate close-to-optimal schedules in polynomial time. The offloading execution managers in CASINO take care of executing jobs according to these schedules. The evaluation shows that CASINO can generate schedules with low completion time, and the execution of computation offloading is done with low overhead. In the future, we plan to enhance CASINO with machine learning algorithms to predict the expected execution times and communication costs of previously unseen jobs.

VIII. Acknowledgment

This research was supported by the NSF under Grants No. CNS 1409523, SHF 1617749, and DGE 1565478, and by DARPA/AFRL under Contract No. A8650-15-C-7521. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, and AFRL.

References

[1] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile Distributed Computing in the Cloud," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, mar 2015, pp. 151–156.

[2] X. Sun and N. Ansari, "Green cloudlet network: A sustainable platform for mobile cloud computing," *IEEE Transactions on Cloud Computing*, 2017.

[3] 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 *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*. New York, New York, USA: ACM Press, 2010, p. 49.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution between Mobile Device and Cloud," in *Proceedings of the sixth conference on Computer systems - EuroSys '11*. New York, New York, USA: ACM Press, 2011, p. 301.

[5] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*. IEEE, mar 2012, pp. 945–953.

[6] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A Computation Offloading Framework for Smartphones." Springer, Berlin, Heidelberg, 2012, pp. 59–79.

[7] A. Bhattacharya and P. De, "A survey of adaptation techniques in computation offloading," 2017.

[8] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," *Handbooks in Operations Research and Management Science*, vol. 4, pp. 445 – 522, 1993, logistics of Production and Inventory.

[9] J. Li, Z. Peng, B. Xiao, and Y. Hua, "Make smartphones last a day: Pre-processing based computer vision application offloading," in *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, June 2015, pp. 462–470.

[10] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems - SenSys '15*. New York, New York, USA: ACM Press, 2015, pp. 155–168.

[11] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa," in *Proceedings of the 9th international conference on Mobile systems, applications, and services - MobiSys '11*. New York, New York, USA: ACM Press, 2011, p. 43.

[12] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim, "Architecture-aware automatic computation offload for native applications," in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*. New York, New York, USA: ACM Press, 2015, pp. 521–532.

[13] W. Zhang, Y. Wen, and H.-H. Chen, "Toward transcoding as a service: energy-efficient offloading policy for green mobile cloud," *IEEE Network*, vol. 28, no. 6, pp. 67–73, nov 2014.

[14] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS : Computation Offloading as a Service for Mobile Devices," *Proceedings of MobiHoc*, 2014.

[15] A. Yousafzai, A. Gani, R. M. Noor, A. Naveed, R. W. Ahmad, and V. Chang, "Computational offloading mechanism for native and android runtime based mobile applications," *Journal of Systems and Software*, vol. 121, no. Supplement C, pp. 28 – 39, 2016.

[16] L. Xiang, S. Ye, Y. Feng, B. Li, and B. Li, "Ready, Set, Go: Coalesced offloading from mobile devices to the cloud," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. IEEE, apr 2014, pp. 2373–2381.

[17] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "Machine learning-based runtime scheduler for mobile offloading framework," *Proceedings - 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013*, pp. 17–25, 2013.

[18] H. Wu and D. Huang, "Modeling multi-factor multi-site risk-based offloading for mobile cloud computing," in *10th International Conference on Network and Service Management (CNSM) and Workshop*. IEEE, nov 2014, pp. 230–235.

[19] K. Liu, J. Peng, H. Li, X. Zhang, and W. Liu, "Multi-device task offloading with time-constraints for energy efficiency in mobile cloud computing," *Future Generation Computer Systems*, vol. 64, 2016.

[20] E. Meskar, T. D. Todd, D. Zhao, and G. Karakostas, "Energy Aware Offloading for Competing Users on a Shared Communication Channel," *IEEE Transactions on Mobile Computing*, vol. 16, no. 1, pp. 87–96, 2017.

[21] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, oct 2016.

[22] M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Moitree: A Middleware for Cloud-Assisted Mobile Distributed Apps," in *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, mar 2016, pp. 21–30.

[23] "Dependency analyzer," http://www.dependency-analyzer.org/ (Accessed:June-10-2017).

[24] "Kryonet," https://github.com/EsotericSoftware/kryonet (Accessed:June-10-2017).

[25] "Battery historian : Android battery profiling tool," https://github.com/google/battery-historian (Accessed:June-10-2017).

[26] "Network monitor : Android network monitoring tool," https://developer.android.com/studio/profile/am-network.html (Accessed:June-10-2017).

[27] "Memory profiler : Android memory analyzing tool and hprof viewer," https://developer.android.com/studio/profile/am-hprof.html (Accessed:June-10-2017).

[28] "Aspectj," https://www.eclipse.org/aspectj (Accessed:June-10-2017).

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97Object-oriented programming*. Springer, 1997, pp. 220–242.

[30] "Performance comparison of java and native c code in android," http://www.learnopengles.com/a-performance-comparison-between-java-and-c-on-the-nexus-5/ (Accessed:June-10-2017).