Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 11-029

Mobilizing the Cloud: Enabling Multi-User Mobile Outsourcing in the Cloud

Chonglei Mei, Daniel Taylor, Chenyu Wang, Abhishek Chandra, and Jon Weissman

November 21, 2011

Mobilizing the Cloud: Enabling Multi-User Mobile Outsourcing in the Cloud

Chonglei Mei, Daniel Taylor, Chenyu Wang, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{chomei,taylor,chwang,chandra,jon}@cs.umn.edu

***Abstract*—Mobile devices, such as smartphones and tablets, are becoming the universal interface to online services and applications. However, such devices have limited computational power and battery life, which limits their ability to execute rich, resource-intensive applications. Mobile computation outsourcing to external resources has been proposed as a technique to alleviate this problem. Most existing work on mobile outsourcing has focused either on single application optimization, or outsourcing to fixed, local resources, with the assumption that wide-area latency is prohibitively high. In this paper, we present the design and implementation of an Android/Amazon EC2-based mobile application outsourcing platform, leveraging the cloud for scalability, elasticity, and multi-user code/data sharing. Using this platform, we empirically demonstrate that the cloud is not only feasible but desirable as an offloading platform for latency-tolerant applications despite wide-area latencies. Our platform is designed to dynamically scale to support a large number of mobile users concurrently by utilizing the elastic provisioning capabilities of the cloud, as well as by allowing reuse of common code components across multiple users. Additionally, we have developed techniques for detecting data sharing across multiple applications, and proposed novel scheduling algorithms that exploit such data sharing for better scalability and user performance. Experiments with our offloading platform show that our proposed techniques and algorithms substantially improve application performance, while achieving high efficiency in terms of resource and network usage.**

I. Introduction

Today, mobile devices such as smartphones and tablets, have become indispensable in our daily lives. With their growing popularity, users have come to rely on them as their go-to devices, and as such expect the features and performance befitting a primary computing device. However, meeting such expectations is challenging for several reasons. First, current battery technology can only support limited computational power in such a portable and lightweight package. Second, mobile devices have neither the processing power nor the memory of traditional desktops and laptops. This presents another bottleneck in the devices' abilities to execute rich, resource-intensive user applications.

One technique that has been proposed to solve these problems is to introduce external computing resources [1], [2]: the resource-intensive portions of applications are split from the main code and delegated for remote execution. There are largely two options for the choice of external resources: (1) local, fixed resources, such as a group of servers, or (2) third party providers, such as the cloud. Using fixed, local resources can work well for low demand outsourcing and has the added benefit of lower latency. It is, however, not ideal when outsourcing demand grows, since the computation power is fixed. On the other hand, the cloud can scale with demand, and commercial cloud platforms, such as Amazon EC2, make implementation of this dynamic scaling simple. However, the conventional wisdom is that wide-area latency is unacceptable for mobile applications. We believe that there is a large class of mobile applications that can tolerate wide-area latency. In fact, even extremely latency-sensitive applications, e.g., Web browsing, are now being hosted in the cloud (Amazon Silk [3]).

It is clear that introducing external resources to handle mobile computations could yield a performance benefit, however there are additional opportunities for increasing performance further. In many instances, the same code components are accessed by multiple users running the same applications, and the same data is shared between multiple applications. In such situations, common code components can be reused and shared data can be cached on the remote platform, saving communication overhead and network traffic associated with transferring the same data. For mobile offloading, the time and energy spent in communication is a large cost, thus this kind of optimization can yield significant performance gains.

In this paper, we present the design and implementation of an outsourcing framework using Amazon EC2 cloud platform to examine the full benefits of cloud-based outsourcing and assess its feasibility in the presence of wide-area latencies. Our cloud platform supports several capabilities which go beyond traditional outsourcing platforms based on local and/or static resources. First, it can dynamically adjust to varying user demand through elastic resource provisioning. Second, it supports intra-cloud sharing of data which significantly reduces communication overhead with the mobile device. Finally, our framework includes a novel scheduler that allocates offload requests to virtual machines in the cloud based on a variety of factors including load, type of task, and predicted data sharing.

The main contributions of this paper are the following:

- **Wide-area outsourcing**: We empirically demonstrate that the cloud is not only feasible but desirable as an offloading platform for latency-tolerant applications despite wide-area latencies, achieving over 90% improvement in runtime and 9.28-fold energy savings over local computation on the mobile device for a compute-intensive image processing application.

- **Dynamic, scalable multi-user offloading platform**: We design and implement an Android/Amazon EC2-based mobile-to-cloud offloading platform which can dynamically scale to support a large number of mobile users concurrently by utilizing the elastic provisioning capabilities of the cloud and by allowing reuse of common code components across multiple users.
- **Multi-application data sharing**: We propose techniques for detecting data sharing across multiple applications, and develop novel scheduling algorithms that exploit such data sharing for better scalability and user performance. Experiments on our offloading platform show that our algorithms can provide up to a 50% reduction in network overhead and a 55% reduction in runtime when compared with a scheduling algorithm does not exploit sharing.

II. Background and Related Work

There are two major approaches for offloading computation from mobile devices to the external resources. The first approach is to partition the application and outsource part of the code to the remote servers based on available resources, such as network availability, bandwidth, and latency. The applications are either partitioned statically or dynamically. Spectra[4] pre-defines different execution plans and selects the best one according to different fidelities at runtime. Calling the Cloud[5] and Wishbone[6] model the mobile application as a consumption graph and dataflow graph respectively. They partition the mobile application statically before execution. Other compiler-assisted approaches [7], [8], [9] and dynamic partitioning approaches [2] have also been proposed. The second approach is to migrate the entire application process[10] or VM[11], [1] through live virtual machine migration[12]. Zap enables process migration with the support of OS. CloneCloud[11] migrates virtual machines in the mobile environment to ease computation outsourcing. It is based on the idea of full VM cloning of a mobile image to a remote server at a coarse-grain level. Cloudlets [1] propose to use local servers as a cloud target, where both the cloudlet server and mobile device run virtual machines.

Offloading mobile computation to local servers has been proposed by different projects[9], [2], [4]. These external resources were located close to the mobile device (e.g. one hop away), under the assumption that distant offloading machines would introduce unacceptable latency. Data staging [13] proposed opportunistic use of untrusted and unmanaged surrogate servers as staging servers for the applications's replicas. [9] outsourced the mobile computation to surrogate servers. MAUI[2] utilizes local resources due to energy concerns. [1] also pointed out the huge energy benefits of offloading code to nearby servers.

Regarding offloading decisions, different metrics have been considered for optimization in prior work. Some studies [14], [15] have analyzed the energy-savings for different communication and computation patterns. In [16], bandwidth and memory are used as the decision-making factors, but CPU and battery life are not considered. Wishbone [6] proposes an approach to partition mobile sensor applications based on network and CPU, but not battery levels.

Overall, most existing work is limited either to static mapping decisions, single application optimization, or outsourcing to fixed local resources. In contrast, our work considers both dynamic wide-area outsourcing as well as multiple applications and users.

III. Feasibility and Challenge of Wide-area Outsourcing

In this paper, we assume that the application has already been partitioned into components eligible for outsourcing. As mentioned in Related Work, such components can be identified by user-annotation, compiler, and runtime techniques. We focus on scheduling methods that can effectively assign such mobile components to remote resources.

To assess the feasibility of wide-area outsourcing, we have developed both cloud and mobile versions of a range of application components in diverse areas such as image processing, speech, and interactive drawing. We modified each mobile application to use either local or remote components for computationally-intensive computations. The details of application modifications and the outsourcing decision process can be found here [17]. In this section, we assess the benefits of outsourcing in a wide range of scenarios.

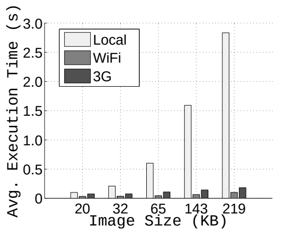
If an outsourcing decision is made, the mobile client sends the input data to the remote server, and receives the processed result (the application components are pre-deployed at the server). Two applications are used to demonstrate the feasibility of wide-area outsourcing: image processing and face detection. Image processing is an application which can apply different effect filters to an image, such as blurring, sharpening, and so on. Face detection is implemented with Haar-Classifier algorithm.

The experimental setting is as follows. The mobile client is an HTC Hero (528 MHz CPU, 200MB RAM, running Android 2.1), and the offloading server is a small instance on Amazon EC2 [18] (1 EC2 Compute Unit and 1.7GB memory)[1]. The average network latency to access EC2 over WiFi and Sprint 3G were measured to be 82ms and 151ms, respectively (compared to 44ms from a wired machine). The application's offloading performance is profiled with both WiFi and 3G networks. In the following sections, *Local* is used to denote that the computation is performed locally on the mobile device, while *Remote-WiFi* and *Remote-3G* denote the remote execution over WiFi and 3G, respectively.

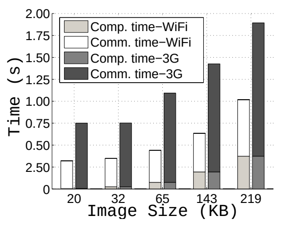
A. Feasibility of Wide-Area Outsourcing

1) Outsourcing Computationally-Intensive Operations: We select the image processing routine blur filter (SimpleBlur) as a computationally-intensive exemplar. The performance

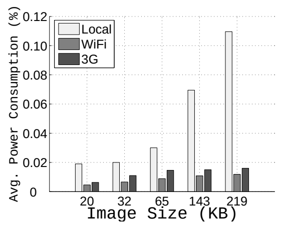
[1]Since the goal of the experiments here is to examine the feasibility of outsourcing in the presence of wide-area latencies, for the experiments in this section, we use only a single EC2 instance as the remote server, and do not employ the dynamic scaling features of EC2.



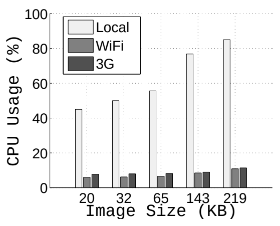
(a) Avg. execution time



(b) Remote execution time breakup



(c) Avg. power consumption



(d) Mobile CPU usage

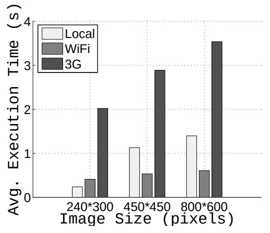
Figure 1: SimpleBlur Results

results are shown in Figure 1. Figure 1a shows the average execution time per run. The remote execution time shown is end-to-end, including both the network communication time and computation time. For all image sizes, local processing is the slowest in every case, while offloading over WiFi is the fastest (achieving about 67.8-96.4% speedup over local processing across the different image sizes). The performance difference of about 60% between the WiFi offloading and 3G is accounted for by the added cost of 3G communication. The compute vs. communication time break down for the remote execution modes is shown in Figure 1b. Here it can be seen that the remote execution time is dominated by the communication time. Additionally, it can be seen that the performance gains increase with the image size. When offloading a 219 KB image, the largest performance gain was 96.4%, offloaded via WiFi.

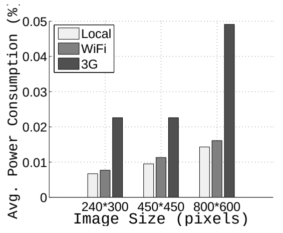
Figure 1c shows the average power consumption, as a percentage of the total battery capacity, used per run. The results mirror those for the end-to-end computation time, with local processing consuming the most power, while offloading via WiFi being most energy-efficient. For instance, when the image size is 219KB, local execution consumes 9.28 times the power of Remote-WiFi, a result which can be attributed to the high computation requirement of the Blur filter, as shown in Figure 1d. It can be seen that when the blurring is executed locally, the mobile CPU usage is over 50%, and increases further with image size. When blurring is performed remotely, the CPU usage drops to below 10%. For each image size, all three execution modes consume nearly the same amount of memory; this is due to the mobile requiring buffers to be maintained, regardless of where execution happens.

These results show that compute-intensive applications can benefit in terms of both performance and energy usage with wide-area outsourcing.

2) Outsourcing Moderate-Compute-Intensive Operations: We selected face recognition as the exemplar of a moderate-compute-intensive operation. The CPU and memory usage have similar patterns to the blur filter (though it runs in less time), thus the details are omitted here. The relation between execution time and power consumption, however, demonstrates different patterns, as shown in Figure 2. For all file sizes, Remote-3G takes more time (Figure 2a) and consumes more power (Figure 2b) than the Local mode.



(a) Avg. Execution Time



(b) Avg. Power Consumption

Figure 2: Face Detection Results

However, there is a tradeoff between Local and Remote-WiFi modes in terms of energy and performance depending on the image size. When the image size is larger than 240×300, although Remote-WiFi consumes about 13% more power, it can achieve $1.47\times$ speedup for the large image.

These results show that there is a potential performance benefit to be had by offloading moderate-compute-intensive applications in a wide-area context, though there are tradeoffs between performance and power consumption that depend on the relative computation-to-communication overheads.

Overall, the above analysis of offloading performance for different application patterns shows that *wide-area outsourcing is not only feasible, but also desirable for compute-intensive applications, with potential performance as well as power savings.*

B. Challenges for High Demand Outsourcing

In the experiments above, we demonstrated the benefits of outsourcing a single application to a static wide-area resource. In each experiment, the number of outsourcing requests is low and the offloading server is able to handle all requests efficiently. This section explores the limitations of such a fixed-resource, wide-area, offloading system under high demand. This can occur when several offload requests are sent in rapid succession, e.g., a mobile user wishing to process a large set of photographs, or multiple users concurrently trying to outsource their computations.

Poor scalability due to static resource allocation: We use the same experimental setting as before. In this experiment, however, four different image operations were performed sequentially on a single image: blur, sharpen, edge-detect, and

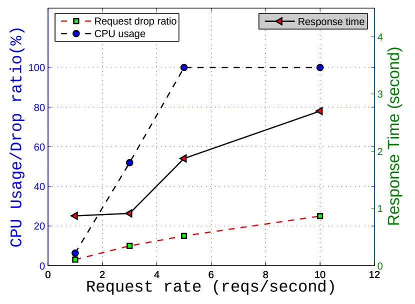


Figure 3: Degradation in server performance under load

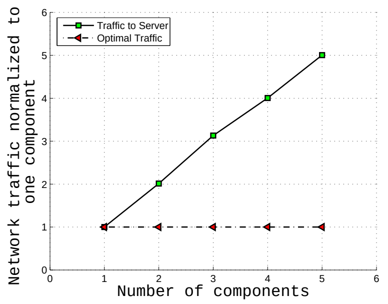


Figure 4: Network overhead due to lack of data sharing

sphere. All of these operations are computationally-intensive. To measure the performance of the server, this sequence was repeated 100 times, for a total of 400 sequential requests. The request rate was tested at 1, 3, 5, and 10 reqs/sec, as shown in Figure 3. We recorded the average CPU usage of the server, the number of requests dropped by the server, and the average response time. The results show that when the request rate is low, between 1 and 3 reqs/sec, the average response time is relatively stable at around 900ms, and the failed request rate is under 10%. After the request rate is increased to 5 reqs/sec, the CPU usage is increased to 100%, 20% of requests are dropped, and the average response is increased to 2 seconds. This overload and performance degradation is inevitable when using a fixed resource pool.

As an additional problem to the fixed-resource model, the usage profiles of many applications are highly variable, and change with user interest, the time of day, and other factors, all of which require a high elasticity on the backend system. Static allocation of resources could result in poor availability during times of high demand and resources sitting idle during low demand periods.

Network overhead due to lack of data sharing: Another issue raised by this experiment is that of network traffic and associated communication overhead. Since the four operations modify the same input image, the source image only needs be sent to the remote resource once, but without knowledge of this data sharing, it would be sent back-and-forth between the server and the mobile multiple times. Figure 4 shows how this discrepancy grows as more components/operations are included. As mentioned above, the outsourcing performance is dominated by the communication cost due to the wide-area nature of the offloading platform, so that a significant reduction in communication time would result in a significant improvement in overall performance.

Since mobile users tend to gravitate to a small number of favored applications and this set may be common across large sets of users, there is a strong possibility that the same outsourcing requests may be issued by many mobile users. In addition, as we scale up to many users and applications, a large amount of external resources are needed. For these reasons, the cloud is a perfect target for outsourcing[2]. A common shared platform like the cloud will enable users to share common application components in the cloud. In addition, the elastic and scalable nature of cloud allocation can provision VMs based upon application component demand. Finally, the cloud can help us exact and exploit data sharing relationships across different outsourcing requests.

IV. Mobile-to-Cloud Offloading Platform

We now present the design and implementation of a cloud-based offloading platform that enables dynamic resource provisioning and achieves high performance by computation co-location. The design of this system is derived from the insights gained from the feasibility study as well as the challenges outlined above. Any mobile device can utilize this offloading system if the mobile application is implemented with the predefined communication interface to the backend system and has identified components for potential outsourcing. Further, we assume that separate cloud versions (ie. non-Dalvik) of these components are provided.

Our current implementation of the offloading system is based on Amazon EC2 as shown in Figure 5. There are two parts in the system: the backend server system and the offloading client on the mobile device, which we describe next. We emphasize that our current implementation is designed to evaluate the efficacy of a cloud-based outsourcing platform, and to explore different optimization opportunities enabled by the cloud (discussed in detail in Section V), and implementation issues such as identifying components or automatic code generation for outsourcing are beyond the scope of this paper as they are well-addressed in Related Work.

A. *Offloading Client*

On the mobile device, we have developed an outsourcing application called ServerTracker to assist in computation outsourcing. This application has two major roles. First, it stores the offloading server's address for each offloaded computation. The server address is assigned to the device by the cloud, and is updated automatically if a new offloading server is

[2]The issue of monetization of cloud resources is outside the scope of this paper. Using external resources whether local or remote always incurs additional cost. We presume that the cost is low enough relative to the benefit obtained to make cloud usage attractive. By aggregating users across cloud servers, we believe the cost will be manageable.

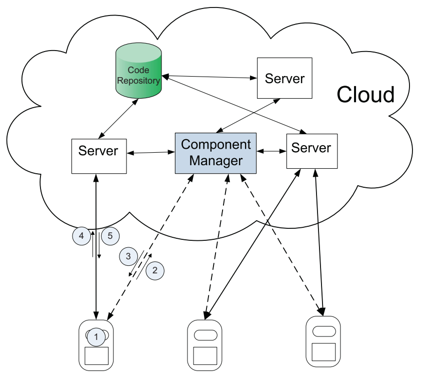


Figure 5: Offloading Backend Design

assigned. As a result, the offloading server can be reassigned by the backend system transparently to the mobile user. Second, the ServerTracker maintains a history of the remote processing times for past remote executions and monitors the current network state, including available network availability, bandwidth, and latency. With the processing history and current network state, ServerTracker estimates the time remote processing will take for a given offloading request, and uses this estimation to make an intelligent offloading decision, whether to execute the request locally or remotely. In this paper, we focus on the case when such an offloading decision has been made.

B. Offloading Backend System

There are three major components in the backend (cloud-side) of the offloading platform: the code repository, the offloading server(s), and the Component Manager. In the following sections, a *computation component* refers to the (outsourced) portion of a mobile application that is executed on the cloud. Since our outsourcing framework is designed for an Android mobile platform, these code components are Java class files.

Code Repository: Each mobile application using the offloading system must have a corresponding cloud component which can perform the resource-intensive computations on behalf of the mobile device. The code repository stores a collection of these code components to be distributed to various offload servers as the need arises. Unlike a static cloud allocation with pre-deployed services, these components are loaded on demand for reasons of cost and efficiency.

Offloading Server: An offloading server does the actual computation work for a mobile device. In our implementation, each offloading server is a virtual machine (VM) running in the cloud that can be easily started, terminated, and merged with other VMs. Upon receiving an offloading request from a client, the offloading server retrieves code from the code repository as needed, and carries out computations for the client.

Component Manager (CM): The Component Manager is the kernel of the backend system which performs the management and scheduling tasks for all incoming offloading requests. The CM is in charge of offload server management (starting, merging, and terminating) and computation assignment. In our implementation, we have partitioned the Component Manager into multiple servers for scalability.

C. Computation Offloading Procedure

A typical computation offloading from the client to the backend happens in the following sequence (See Figure 5):

Step 1: Given a mobile application, the ServerTracker on the offloading client decides if the current network conditions are condusive to offloading a task.

Step 2: If a remote execution decision is made, the offloading client sends an offloading request to the Component Manager.

Step 3: The Component Manager selects an appropriate server (VM) from the existing offloading servers using its scheduling algorithm (discussed in Section V). It returns the IP address of the selected server to the offloading client.

Step 4: After the offloading client receives the address of the offloading server, the ServerTracker records the server address locally. The application then sends the input data (if the data is not already available in the cloud) and the operation to be executed to the offloading server.

Step 5: After the offloading server receives the data package, it parses it to determine the computation to be performed. If the required code components are not available locally on the server, it downloads them from the code repository. Then it carries out the computation and sends back the result to the mobile client.

V. Computation Offload Request Scheduling

We now describe how offloading requests are scheduled to offloading servers. When a computation offload request arrives to the cloud, the Component Manager needs to identify the "best" offloading server to execute that request based on the location of the already placed computation components and the server resource utilization states. An ideal scheduling algorithm would achieve load balance and high resource utilization across the offloading servers, while providing high performance to the mobile users. There are two main criteria that must be considered by the scheduling algorithm to achieve these goals:

- When to dynamically provision offloading servers?
- Where to place the offloaded computation components?

These criteria are intended to overcome the challenges discussed in Section III-B: poor scalability due to static resource allocation, and network overhead due to lack of data sharing among application components, respectively.

A. Dynamic Provisioning

To achieve scalability and efficiency, the Component Manager decides when to create, destroy, or merge servers (VMs). To do this, it periodically collects each offloading server's utilization states and makes scheduling decisions accordingly. In our current implementation, only the server's CPU usage is used to assess the utilization state, which can be extended to

Algorithm 1 Dynamic Provisioning

```
1: Input: $servers$
2: if all $server.cpuUsage > CPUHighThreshold$ then
3:     $spawnNewServer()$
4: else
5:     for $s$ in $servers$ do
6:         if $s.cpuUsage < CPULowThreshold$ then
7:             $Candidates$ = set of all servers with $cpuUsage < CPUHighThreshold$
8:             $targetServer = server$ in $Candidates$ and $server$ with $lowestCPU$ and $server != s$
9:             if $targetServer != null$ then
10:                $merge(s, targetServer)$
11:            end if
12:        end if
13:    end for
14: end if
```

other resources such as memory, network I/O and Disk I/O. We set a High and a Low threshold on the server's CPU usage to indicate overload and underload conditions respectively. The dynamic provisioning algorithm is shown in Algorithm 1, and consists of two main operations:

1) Server Creation: At each interval when Component Manager collects the offloading servers' CPU utilization values, it checks to see if they are overloaded, by comparing the utilization value against the High threshold. If all servers are seen to be overloaded for a sustained period (for last 3 intervals in our implementation), a new server is spawned. In addition, the system always maintains a spare offloading server in the system to avoid server startup overhead, since it takes around 60 seconds to boot a new instance in EC2.

2) Server Merging: Component Manager identifies underloaded servers by comparing their utilization against the Low threshold, and tries to merge them with other servers for scalability. For each underloaded (source) server, it identifies a target server for merging as one below the High threshold with the least load. To merge two offloading servers, the two servers' computation states are preserved by copying the intermediate data files from the source server to the target server. After the two servers have been merged, the Component Manager sends a message containing the new target server's IP address to all mobile clients using components on the source server. In our design, the merging process is transparent to the mobile user.

B. Impact of Component Location on Performance

For the dynamic provisioning mechanisms outlined above, we have implicitly assumed each outsourcing request to be independent, so that it can be executed on any server. However, in practice, multiple applications may need the same computation components or may share the same data for their computation, so there may be dependencies between different requests. For example, the accuracy of face detection is affected by the image quality, which can be improved with image processing. In this case, the user may first invoke the image processing application to preprocess an image containing a face, and then invoke the face detection application with the output of the preprocessing operation. Thus, upon receiving an outsourcing request, the scheduler must consider such sharing opportunities while determining the location to place the computation component.

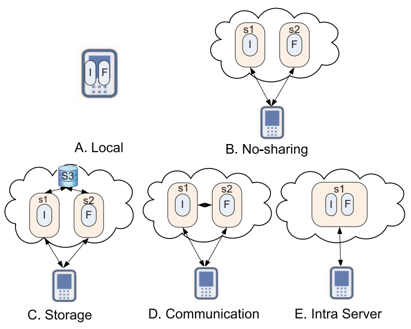


Figure 6: Different data sharing approaches

In the above example, there are different approaches for sharing intermediate data (an image, in this case) between the two applications. Figure 6 shows the five potential scenarios to share the intermediate data. In each scenario, **F** and **I** stand for face detection and image processing respectively. In addition, S1 and S2 stand for two offloading servers that could host the compute components.

- *No offloading (Local):* This corresponds to the local (non-offloading) execution of both face detection and image processing on the mobile device, where the data is shared.
- *No sharing in the cloud (No-sharing):* In this scenario, computation is outsourced for both image processing and face detection, which are executed on servers S1 and S2 respectively, with no cloud sharing of intermediate data. The sharing has to be done via the mobile device, which involves sending the intermediate data back and forth between the mobile and the cloud.
- *Sharing via backend storage (Storage):* In this scenario, all data is stored on a persistent backend storage, such as the Amazon Simple Storage Service (S3). The server S2 (hosting face detection) can then download the intermediate image from Amazon S3 directly, so that the data sharing is intra-cloud.
- *Direct communication (Communication):* In this case, the intermediate data is stored locally on server S1. When face detection is invoked, S2 fetches the intermediate data from S1 through a direct network connection.
- *Intra-server sharing (Intra-server)*: In this case, both face detection and image processing applications are hosted on the same offloading server, so that face detection can access the intermediate data locally on the server from its file system/memory cache.

First, we examine which of these choices would be well-suited to maximize outsourcing performance in the presence of sharing. We conduct an experiment to compare performance of these different approaches with a medium-size (405$\times$405) and a large-size (800$\times$600) image, with offloading performed over WiFi. In each scenario, image processing is performed

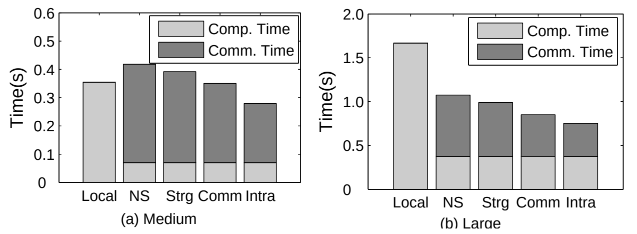


Figure 7: Processing time for different data sharing approaches

on S1 and it caches the intermediate image which is used by face detection. As the image processing is the same for each scenario, its performance results are omitted, and we only show performance for face detection, including the total communication and computation time.

Figure 7 shows the processing time for the different scenarios. The first thing to note is that the communication time dominates the remote processing time. Therefore, outsourcing performance depends on the location of the computation components and the communication link between them. For a large image size, the remote processing always has better performance than local. In particular, we observe that intra-server sharing has the best remote processing performance (54.9% better than the local computation). Even for a smaller image size, where computation is less dominant, the intra-server sharing outperforms local processing by 21.4% while all other remote processing approaches lead to worse performance. The reason is that computation components **F** and **I** are collocated on the same server, which eliminates the communication cost between offloading servers, backend storage, or the mobile device, needed for the other scenarios. Note that this collocation can expand the range of requests that can benefit from outsourcing.

Overall, these results show that *co-locating components that share data can result in improving application performance substantially by reducing redundant communication overhead.*

C. Component Placement Strategies

Based on the insights gained above, we now present three different component placement strategies with the goal of reducing communication overhead. We assume that the Component Manager does not explicitly know which code components will share data, though as we will discuss, it may attempt to infer this information in an application-transparent manner. For the scope of this work, we only consider data sharing between applications executed by the same user, and defer cross-user data sharing as part of future work. However, we do allow compute component reuse between multiple users executing the same applications.

1) User-Centric: In this approach, the system assigns each offloading server to only one mobile user, so that each mobile user's requests always go to the same server. If the server is overloaded, a new server is created and assigned to the same user, based on the dynamic scheduling criterion. The algorithm is described in Algorithm 2. The intuition behind

Algorithm 2 User-centric

```
1: Input: RequestForComponent $c$, User $u$
2: Output: $bestServer$
3: $Candidates =$ set of all servers that belong to $u$
4: $bestServer = server$ in $Candidates$ and $server$ with $lowestCPU$
```

Algorithm 3 App-centric

```
1: Input: RequestForComponent $c$
2: Output: $bestServer$
3: $servers$= set of all servers that host $c$ where $server.cpuUsage < CPUHighThreshold$
4: if $servers! = null$ then
5:     $bestServer = server$ with $lowestCPU$
6: else
7:     $bestServer = server$ with $lowestCPU$ and $lowestCPU < CPUHighThreshold$
8:     if $bestServer == null$ then
9:         $bestServer = spawnNewServer()$
10:    end if
11: end if
```

this approach is that if one of the user's components needs to access data from another component, it can be obtained locally. The downside of this approach is low utilization and high cost.

2) App-Centric: In this approach, each server hosts a subset of application components and user requests can be mapped to any offloading server that holds the required component(s). When a new request arrives, the system tries to assign it to the least loaded offloading server that contains the requested component. If it fails, based on the dynamic scheduling criterion, the request is assigned to the offloading server with the lowest CPU usage, or if no offloading server is available, a new server is spawned and the required components are downloaded for execution. The algorithm is shown in Algorithm 3.

This approach allows component reuse across multiple users executing the same applications, and hence, can yield high utilization and requires fewer number of servers, since a new offloading server will be created for a component only when server load is high on all servers at the backend. However, since different components belonging to the same user could be hosted by different offloading servers, it increases the likelihood that components that share data are hosted on different offloading servers, resulting in higher communication overhead, as discussed before.

3) Co-location: To achieve benefits of both the user- as well as app-centric approaches, we propose a hybrid technique called *Co-location*, which enables the reuse of common components among multiple users like the App-centric approach, while also identifying and collocating shared components like the User-centric approach.

Co-location attempts to predict which components will share data based on their temporal locality of access by a user. The intuition behind this approach is that if a mobile user often tries to access two (or more) components within a short period of time, then there is a high probability that the user may be sharing data across these components.

To detect temporal relationships, we use a well-known

Algorithm 4 Co-location

```
1:  **Input:** RequestForComponent $c$
2:  **Output:** $bestServer$
3:  $bestServer = null$
4:  **if** $c$ in any AssociationGroup **then**
5:      $group$ = the AssociationGroup with highest support
6:      $servers$ = set of servers that contain members of $group$ and
        $server.cpuUsage < CPUHighThreshold$
7:      **if** $servers! = null$ **then**
8:          $bestServer$ = the server with $lowestCPU$ in $servers$
9:      **end if**
10: **end if**
11: **if** $c$ not in any AssociationGroup or $bestServer == null$ **then**
12:     $bestServer$=App-centric(c)
13: **end if**
```

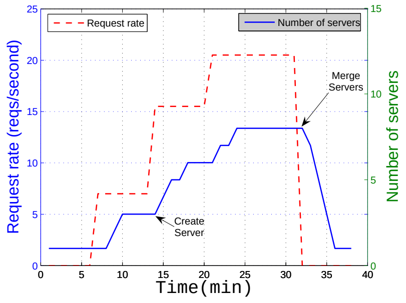


Figure 8: Scalability of the Offloading System

data mining technique known as *association analysis* [19]. *Association analysis* is derived from market basket transaction processing which intends to discover the hidden relationships between different items. The uncovered relationships can be represented in the form of association rules or sets of frequent items. The classic example is the association of $\{Diapers\} \rightarrow \{Beer\}$ from the transactions of a store. We use association analysis to determine which computation components are likely to be accessed together. The identified component group, i.e., *Association Group*, is used to guide the computation assignment.

In terms of implementation, the Component Manager logs each time-stamped component access from the mobile device to the offloading system. The association analysis algorithm uses $< user, component, time >$ tuples in the log as the input transactions to identify those components with the potential for co-location. The Component Manager performs association analysis periodically to identify such Association Groups.

The Co-location algorithm is shown in Algorithm 4. When a new computation request arrives, this algorithm first checks whether the requested computation component is in any Association Group. If it belongs to multiple groups, the group with largest support is selected. Then it locates all of the servers that host the members of the selected Association Group, and selects the server with the lowest CPU usage. If the component is not in any Association Group, then it uses the App-centric algorithm to select a server.

VI. Evaluation of Scheduling Algorithms

A. Experimental Setup

As discussed in Section III, our platform consists of Android mobile device clients and Amazon EC2 offloading backend. For the experiments in this section, we host the backend components such as the Component Manager and offloading servers on small and micro instance types respectively. The code repository is hosted on a small instance for efficiency. There are 10 computation components available in the code repository, which are 10 different image processing filters. We generate the request workload by emulating 100 mobile users on one laptop (2.53GHz Duo CPU, 2GB RAM). The requests arrive at fixed time intervals, determined by the request rate. Each user uses a certain number n of components, where n is picked from a normal distribution $N(3,1)$. In each experiment, some of the requests are designated as *sharing requests*—these share data among their components. The percentage of sharing requests in the workload and the total request rate to the system are varied for different experiments.

B. Benefit of Dynamic Provisioning

We first show the benefit of dynamic provisioning (Section V-A) by showing that, in our backend offloading system, the amount of allocated resources scales up and down automatically based on the arrival rate of the computation requests. In our experiments, the High and Low CPU usage thresholds were set as 70% and 30% respectively. Figure 8 plots the number of offloading servers in the backend system with the changing request rate over time. The request rate is 0 at the beginning, and is successively increased to 7, 15.5, and 20.5 reqs/second at time=6 min, 15 min, and 22 min respectively. The last request is sent at at time=31 min, after which request rate goes back to 0. As we can see, the number of offloading servers also varies based on the request rate. Starting from one offloading server (recall there is always a spare server in the system) in the system at time 0, as the request rate increases, the number of offloading servers also increases. It reaches a peak of 8 servers when the request rate is 20.5 reqs/second. There is a delay for the number of servers to become stable after a new one is added due to the 1 min instance creation latency in EC2. At time=31 min, the provisioning algorithm starts to merge offloading servers as their load drops below the Low threshold, finally falling back to 1 server at the end.

The results show that *our system can successfully self-scale its offloading servers according to incoming request load, thus achieving high server utilization.*

C. Comparison of Component Placement Strategies

We now compare the three component placement strategies described in Section V-C: User-centric, App-centric, and Co-location. As a baseline, we also compare them to the No-sharing case (Section V-B) where all intermediate data between requests flows to and from the mobile client. We examine their impact on two metrics: (i) *server utilization* as a measure of backend resource efficiency, and (ii) *network traffic* as a measure of both user performance and network

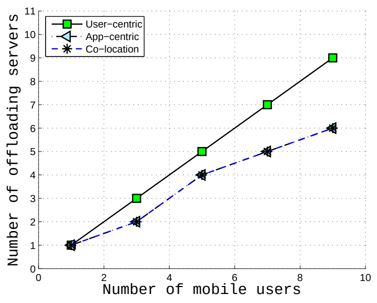


Figure 9: Number of Servers

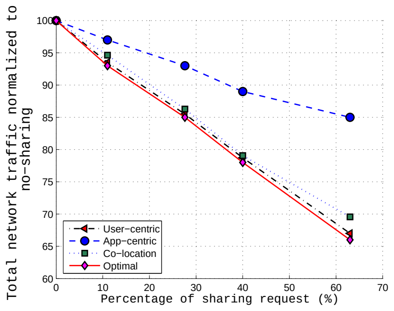


Figure 10: Total Network Traffic Normalized to No-sharing

overhead, since communication is the dominant outsourcing cost, as discussed in Sections III and V.

1) Impact on Server Utilization: Figure 9 shows the number of servers used for the three placement algorithms as the number of mobile users increases. The request rate for each user is 2 reqs/second. When there is only one user, all three algorithms use one offloading server. However, as the number of users increases, the number of servers for User-centric increases linearly. The reason is that User-centric assigns at least one offloading server to each mobile user irrespective of the required computing resources to service that user. On the other hand, both App-centric and Co-location try to “fill up” a server before launching a new one. Therefore, both App-centric and Co-location use fewer servers than User-centric (The curves for App-centric and Co-location overlap). Moreover, the difference in the number of servers used by User-centric vs. App-centric and Co-location increases as the number of users increases.

This result shows that *compared to User-centric, both App-centric and Co-location placement strategies are much more efficient and scalable in the use of server resources as the number of users increases.*

2) Impact on Network Traffic: Figure 10 shows the total network traffic for the three placement algorithms, as we vary the *sharing ratio*: the number of sharing requests as a fraction of the total number of offloading requests. A low/high sharing ratio corresponds to few/many requests sharing data among their components, respectively. In this experiment, each run lasts for 15 minutes and the request rate is fixed at 15.5 reqs/second. The network traffic in the figure is normalized by that for the No-sharing case. The Optimal line is the total amount of network traffic if all the sharing requests were collocated successfully. First, the figure shows that the normalized traffic diminishes with increasing percentage of sharing requests for all algorithms. This is expected since each of our placement algorithms can take advantage of higher sharing opportunities via intra-VM or intra-cloud communication, while No-sharing has to send more redundant traffic to the mobile as more sharing occurs. Secondly, of the three algorithms, User-centric can save the most network traffic, and is close to Optimal. Co-location is only slightly worse than User-centric, and is much better than App-centric. Further, the gap in their traffic increases with increasing sharing ratio.

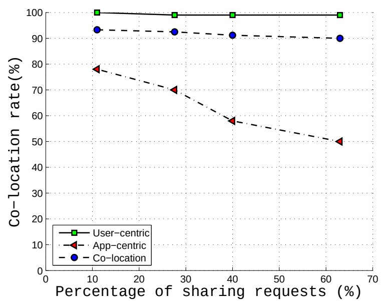


Figure 11: Co-location Rate

The reason for the lower network overhead of User-centric and Co-location is apparent from Figure 11. This figure shows the *co-location rate* of the three algorithms as the ratio of sharing request in the workload is varied. The co-location rate is the percentage of sharing requests whose components are successfully collocated on the same server. A higher co-location rate corresponds to a greater reduction in network traffic and better offloading performance for the application, since most sharing communication takes place intra-server (as discussed in Section V-B). The results show that User-centric can achieve almost 100% co-location rate. The reason is that the same user’s computation is always performed on the same server, and the components are collocated naturally. The co-location rate for Co-location is over 90% in all cases, since Association Analysis is able to identify sharing patterns successfully in most cases. However, the co-location rate for the App-centric approach effectively decreases as the ratio of sharing requests increases, because it makes its placement decision without considering the sharing between components, and hence its placement of components is effectively random.

These results show that *compared to App-centric, User-centric and Co-location placement strategies achieve much higher co-location rates and thus, much lower network traffic.*

Overall, our results above show that while App-centric is the most efficient in terms of server utilization and User-centric has the lowest network overhead, *Co-location achieves the benefits of both these algorithms—it uses the same number* *of servers as App-centric, and can reduce nearly the same amount of network traffic as User-centric.* The actual benefit to the application depends on the amount of network traffic that is reduced. For the face detection case, Co-location provides a 31% performance inprovement compared with No-sharing case (when the image size is 800×600). And for the image processing, it provides a 55% performance when blur filter is used.

3) System Overhead: An overhead for an offloading request is the component preparation cost: the cost to download the needed components from the code repository and load them in the server memory. Note that this cost needs to be paid only when a component is not already available on the server. To quantify the component preparation cost, we implemented two versions of the code repository, residing on Amazon S3 and EBS respectively. We measured the preparation cost for components of varying sizes for three different applications: 8.9KB, 1.4MB (corresponding to image processing), and 61MB respectively. For each case, the server takes an average of 5ms to load the component into memory, however, the download time was 70ms, 397ms, 12110ms and 6ms, 109ms, 2325ms respectively for S3 and EBS for these data sizes. Thus, to reduce the cost of computation component preparation, EBS is a better choice. When User-centric placement is used, each user needs to download all components of their applications from code repository, however, Co-location and App-centric reuse the components for different users. Thus, *App-centric and Co-location can improve the outsourcing performance for users in terms of startup latency as well.*

The Association Analysis algorithm is executed every 3 minutes on the transactions over that period. When the Association Group size is set to 10 components, with 120 total components in 10,000 requests, each run takes around 0.51 seconds, while for our experiments, we used 10 components in 500 requests, which takes only 30ms.

VII. Concluding and Future work

In this paper, we presented the design and implementation of an Android/Amazon EC2-based mobile-to-cloud computation outsourcing platform. Using this platform, we empirically demonstrated that the cloud is not only feasible but desirable as an offloading platform for latency-tolerant applications despite wide-area latencies. We showed how our platform can dynamically scale to support a large number of mobile users concurrently by utilizing the elastic provisioning capabilities of the cloud. We proposed three component placement algorithms: User-centric, App-centric, and Co-location, that allow component reuse and data sharing to varying degrees. The Co-location algorithm uses techniques for detecting data sharing across multiple applications, and our experimental results showed that it achieves resource usage efficiency comparable to the App-centric algorithm, while approaching the User-centric algorithm in its low network overhead. Overall, it was able to provide up to 50% reduction in network overhead and 55% reduction in runtime over a No-sharing algorithm.

The current server provisioning mechanism only focuses on CPU usage, we plan to extend it to other metrics, such as memory, network I/O, and disk I/O. In addition, the current implementation only focuses on data sharing opportunities for the same user. In the future, we plan to explore how this can be extended across multiple users.

References

[1] M. Satyanarayanan, P. Bah, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Journal of IEEE Pervasive Computing*, vol. 8, 2009.

[2] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," *ACM MobiSys*, 2010.

[3] *http://amazonsilk.wordpress.com/*.

[4] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performace, energy, and quality in pervasive computing," *Proceedings of the 22nd International Conference on Disributed Computing Systems*, 2002.

[5] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Middleware 09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[6] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based partitioning for sensornet applications," in *Proceeding of USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[7] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: a partition scheme," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES'01)*. New York, NY, USA: ACM, 2001, pp. 238–246.

[8] C. Wang and Z. Li, "Parametric analysis for adaptive computation offloading," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*. New York, NY: ACM, 2004, pp. 119–130.

[9] S. Kim, H. Rim, and H. Han, "Distributed execution for resource-constrained mobile consumer devices," *IEEE Transactions on Consumer Electronics (TCE)*, pp. 376–384, May 2009.

[10] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *Proceedings of the 5th symposium on Operating Systems Design and Implementation(OSDI)*, 2001.

[11] B.-G. Chun and P. Maniati, "Augmented smartphone applications through clone cloud execution," *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[12] Deshpande, Umesh, Wang, Xiaoshuang, and K. Gopalan, "Live gang migration of virtual machines," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996151

[13] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan, "Data staging on untrusted surrogates," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 15–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1090694.1090697

[14] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[15] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, 2010.

[16] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt, "Adaptive offloading for pervasive computing," *IEEE Pervasive Computing*, vol. 3, pp. 66–73, 2004.

[17] C. Mei, J. Shimek, C. Wang, A. Chandra, and J. Weissman, "Mobile computation offloading framework," Department of Computer Science and Engineering, University of Minnesota, Twin Cities, Tech. Rep. 11-006, Mar. 2011.

[18] *http://aws.amazon.com/ec2/*.

[19] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2006.