Sharing-aware Cloud-based Mobile Outsourcing

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 smart phones 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 resource-intensive applications. Computation outsourcing to external resources has been proposed as a technique to alleviate this problem. Most existing work on mobile outsourcing has focused on either single application optimization or outsourcing to fixed, local resources, with the assumption that wide-area latency is prohibitively high. However, the opportunity of improving the outsourcing performance by utilizing the relation among multiple applications and optimizing the server provisioning is neglected. In this paper, we present the design and implementation of an Android/Amazon EC2-based mobile application outsourcing framework, leveraging the cloud for scalability, elasticity, and multi-user code/data sharing. Using this framework, we empirically demonstrate that the cloud is not only feasible but desirable as an offloading platform for latency-tolerant applications. We have proposed to use data mining techniques to detect data sharing across multiple applications, and developed novel scheduling algorithms that exploit such data sharing for better outsourcing performance. Additionally, our platform is designed to dynamically scale to support a large number of mobile users concurrently. Experiments show that our proposed techniques and algorithms substantially improve application performance, while achieving high efficiency in terms of computation resource and network usage.**

I. INTRODUCTION

Today, mobile devices such as smart phones 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 computers.

One technique that has been proposed to solve these problems is to introduce external computing resources [1], [2]: 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. The conventional wisdom is that wide-area latency is unacceptable for mobile applications, therefore, local servers are the best choice. 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]).

The benefits of introducing external resources to individual application and mobile device have been well studied. However, we are unaware of any work that exploits the relation between different applications to further improve the outsourcing performance. In many instances, the same code components may be accessed by different users running the same or different applications, and the 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 computation outsourcing, 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 benefits of sharing-aware cloud-based mobile computation outsourcing.

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.
- **Multi-application data sharing**: The key contribution is that we propose to use data mining techniques for detecting potential data sharing across multiple applications, and develop novel scheduling algorithms that exploit such data sharing to achieve better outsourcing performance. Experiments 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.
- **Dynamic, scalable multi-user offloading platform**: We 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 capabilities of the cloud.

II. BACKGROUND AND RELATED WORK

Offloading mobile computation to local servers has been proposed by different projects[4], [2], [5]. These external resources were located close to the mobile device (e.g. one

This work was supported by NSF Grant IIS-0916425

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.48

hop away), under the assumption that distant offloading machines would introduce unacceptable latency. Data staging [6] proposed opportunistic use of untrusted and unmanaged surrogate servers as staging servers for the applications's replicas. [4] outsourced the mobile computation to surrogate servers. MAUI[2] utilizes local resources due to energy concerns. Our work, on the other hand, uses the cloud as the outsourcing platform due to its resource-richness despite the presence of wide-area latency. In addition, the current research neglected the fact that by optimizing the usage of servers and utilizing the relation of different applications can reduce network communication overhead and therefore further improve performance.

There are two major approaches for offloading computation from mobile devices to external resources. First, the application is partitioned and part of the code is outsourced based on available resources, such as network availability, bandwidth, and latency[5], [7], [8], [9], [2]. The second approach is to migrate the entire application process[10] or VM[1], [11] through live virtual machine migration. In contrast, our work is primarily focused on the backend optimization, which is independent of the outsourcing technique. We employed a component-based application partitioning technique that can dynamically offload the computation.

Dynamic resource provisioning is one of the main features of the Cloud. Amazon AWS provides Auto Scaling service[1] to allow users to specify a customized condition to automatically scale Amazon EC2 capacity. However, there is a long delay to detect the changed condition. Virtual Machine(VM) collocation techniques have been proposed to reduce communication overhead between different VMs. Xenloop [12] improves the communication performance between controlled Xen VMs through an inter-VM shared memory channel. Starling [13] tries to co-locate VMs that contribute to more communication overhead. Several resource scheduling techniques [14], [15] for high performance and Grid computing have focused on scheduling of resources that match the capacity of compute jobs. However, most of the work assumes that jobs are independent while relations between them are neglected. CloudViews[16] studied the opportunity to share data between different web services in the public Cloud. Our work is focusing on co-locating fine-grained mobile computation components based on data sharing in the Cloud, which is different from the current work. Our dynamic provisioning scheduler is more responsive to changes of resource utilization in the Cloud and considers relations between different jobs.

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, face detection, speech, and interactive drawing. We modified each mobile application to use either local or remote components for the computationally intensive part. The details of application modifications and the outsourcing decision process can be found in [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 computation components are pre-deployed at the server). Two applications are used to demonstrate the feasibility of wide-area outsourcing: image processing and face detection.

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 [2] (1 EC2 Compute Unit and 1.7GB memory). 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). 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.

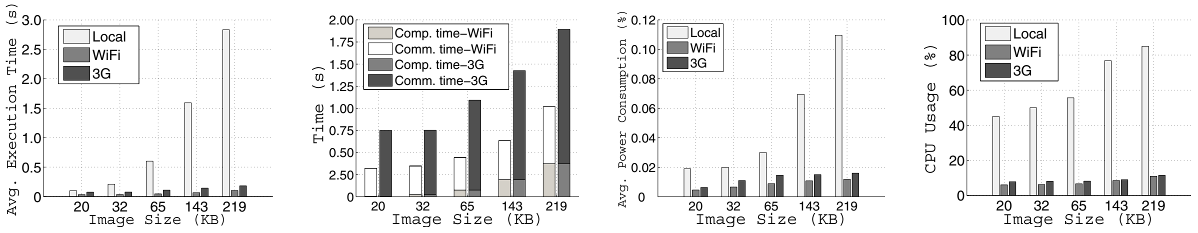
A. Feasibility of Wide-Area Outsourcing

We select the image processing routine blur filter (SimpleBlur) as a computationally-intensive example. The performance 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 and computation time. For all image sizes, local processing is the slowest, while offloading over WiFi is the fastest (achieving about 67.8-96.4% speedup over local processing). The performance difference of about 60% between the WiFi 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.

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 Remote-WiFi being most energy-efficient. For instance, when the image size is 219KB, local execution consumes 9.28 times the power of Remote-WiFi, which can be attributed to the high computation requirement, as shown in Figure 1d. It shows that when SimpleBlur is executed locally, the mobile CPU usage is over 50%, and increases further with image size. When performed remotely, the CPU usage drops to below 10%.

[1]http://aws.amazon.com/autoscaling

[2]http://aws.amazon.com/ec2



(a) Avg. execution time (b) Remote execution time breakup (c) Avg. power consumption (d) Mobile CPU usage

Figure 1: SimpleBlur Results

We also selected face detection as an example of a moderate-compute-intensive application. Due to space limitation, we omit the details here, which can be found in [17]. The results show that there is a potential performance benefit by offloading as well, 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 shows that *wide-area outsourcing is feasible for certain classes of compute-intensive applications, with potential performance gains and 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, where 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 outsourcing their computations.

Network overhead due to lack of data sharing: 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 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 first issue is that of network communication overhead. Since the four operations modify the same input image, the source image needs only be sent once. However, without knowledge of this data sharing, the image would be sent back-and-forth between the server and the mobile device multiple times. Figure 2 shows how this discrepancy grows as more components/operations are included. As mentioned above, the outsourcing performance is dominated by the communication cost, so that a significant reduction in communication time would result in a significant improvement in overall performance.

Poor scalability due to static resource allocation: As there is only one server available in this experiment, we observe that there is significant delay when there are a large amount of

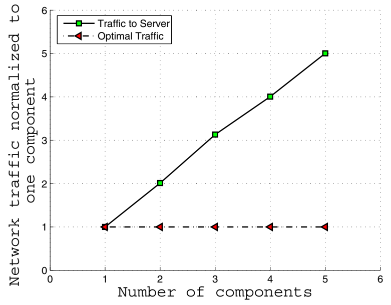


Figure 2: Network overhead due to lack of data sharing

concurrent outsourcing requests. An additional problem is that 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.

As the outsourcing scales 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.[3] A shared platform like the cloud will enable users to share common application components and data. It can help exploit data sharing relations across different outsourcing requests and therefore reduce outsourcing cost.

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. Any mobile device can utilize this offloading system if the mobile application is implemented with the predefined communication interface to the backend system. Further, we assume that separate cloud versions(i.e. non-Dalvik) of the outsourcing components are provided.

Our current implementation is based on Amazon EC2 as shown in Figure 3. There are two parts: the backend server

[3]The issue of monetization of cloud resources is outside the scope of this paper. We presume that the cost is low enough relative to the benefit obtained to make cloud usage attractive.

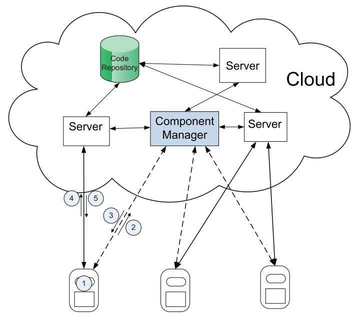


Figure 3: Offloading Backend Design

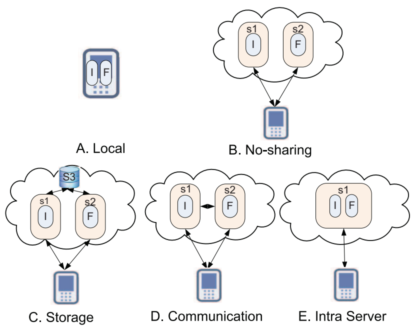


Figure 4: Different data sharing approaches

system and the offloading client on the mobile device. 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). 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 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 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, which are used to make offloading decisions. In this paper, we focus on the case when such offloading decisions have been made, but not how to make the decision.

B. *Offloading Backend System*

There are three major components in the backend (cloud-side): 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.

Code Repository: The code repository stores a large collection of computation components to be distributed to various offload servers as the need arises dynamically.

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, it retrieves code from the code repository as needed, and carries out computations.

Component Manager (CM): The Component Manager is the kernel of the backend system which performs the management for offloading servers, and scheduling tasks for all incoming offloading requests. We have replicated the Component Manager into multiple servers for scalability.

C. *Computation Offloading Procedure*

A typical computation outsourcing from the client to the backend happens in the following sequence (See Figure 3):

Step 1: ServerTracker decides to offload an operation.

Step 2:The offloading client sends an offloading request to the Component Manager.

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

Step 4: The ServerTracker stores the received server address locally. The application then sends the input data (if the data is not already available in the cloud) to the offloading server.

Step 5: The offloading server receives the data package, performs the required computation and sends back the results. If the required computation components are not available locally on the server, it downloads them from the code repository.

V. Computation Offload Request Scheduling

We now describe how offloading requests are scheduled to offloading servers. When a computation offload request reaches 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 ensure high performance for mobile users, while achieving load balance and high resource utilization across the offloading servers. Two main criteria must be considered by the scheduling algorithm:

- Intelligently assign the offloading request
- Dynamically provision offloading servers

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

A. Impact of Component Location on Performance

In Section III-A, we have implicitly assumed each outsourcing request to be independent, 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.

In the above example, there are different approaches for sharing intermediate data (an image, in this case) between the two applications. Figure 4 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 execution of both face detection and image processing on the mobile device, where the data is shared.
- *No sharing in the cloud (No-sharing):* The intermediate data sharing has to be done via the mobile device, which involves sending the intermediate data back and forth.
- *Sharing via backend storage (Storage):* All data is stored and shared through a persistent backend storage, such as the Amazon Simple Storage Service(S3).
- *Direct communication (Communication):* 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)*: 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 (405×405) and a large-size (800×600) image, with offloading performed over WiFi. In each scenario, image processing is performed on S1 and S1 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 both communication and computation time.

Figure 5 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

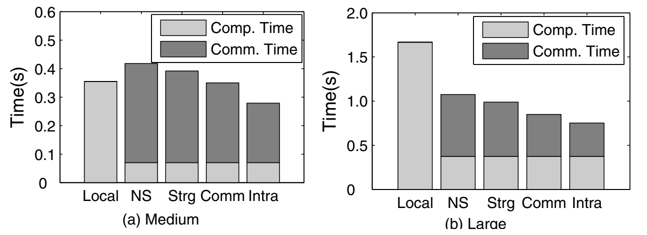


Figure 5: Processing time for different data sharing approaches

a large image size, the remote processing always has better performance. In particular, we observe that intra-server sharing has the best remote processing performance (54.9% better than the local computation). Even for a medium 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 co-located on the same server, which eliminates the communication cost between offloading servers, backend storage, or the mobile device. Note that this co-location 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.*

B. 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 CM 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 and security issues as part of future work. However, we do allow computation component reuse between multiple users outsourcing the same application.

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, which is discussed in V-C. The intuition behind 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, CM 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 for the request.

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 the 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 co-locating 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 data mining technique, i.e., *Sequence Mining* [18]. It tries to find statistically relevant patterns between data examples where the values are delivered in a sequence. We use this technique to determine which computation components are likely to be accessed together as a group. 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 Sequence Mining algorithm is executed periodically and uses $< user, component, time >$ tuples in the log to identify those components with the potential for co-location.

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. It then 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, it then uses the App-centric algorithm to select a server.

C. Dynamic Provisioning

To achieve scalability and efficiency, the Component Manager decides when to create 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 other resources such as memory and 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 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 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. Then the source server is killed.

VI. Evaluation of Scheduling Algorithms

A. Experimental Setup

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 in EC2 respectively. The code repository is hosted on a small instance for efficiency. 10 computation components are 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. Comparison of Component Placement Strategies

We now compare the three component placement strategies described in Section V-B: User-centric, App-centric, and Co-location. As a baseline, we also compare them to the No-sharing case (Section V-A) 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 overhead, since communication is the dominant outsourcing cost.[4]

1) Impact on Server Utilization: Figure 6 shows the trend of number of servers used for the three placement algorithms

[4]Energy consumption is not covered in this section since it is closely related to network communication overhead.

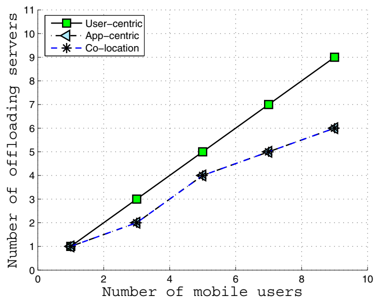


Figure 6: Number of Servers

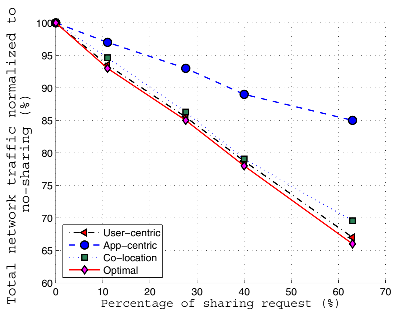


Figure 7: Total Network Traffic Normalized to No-sharing

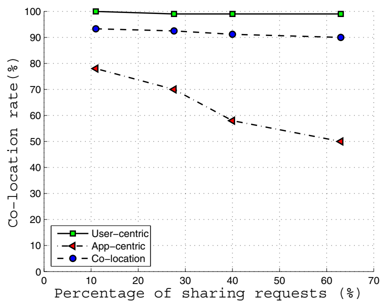


Figure 8: Co-location Rate

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 more efficient and scalable in the use of server resources as the number of users increases.*

2) Impact on Network Traffic: Figure 7 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 co-located 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.

The reason for the lower network overhead of User-centric and Co-location is apparent from Figure 8. 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 co-located 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-A). 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 co-located 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

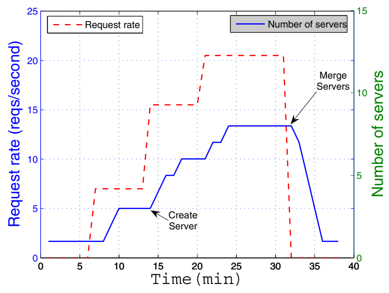


Figure 9: Scalability of the Offloading System

a 31% performance improvement(total execution time) compared with No-sharing case (when the image size is 800×600). And for the image processing, it provides a 55% performance improvement when blur filter is used.

C. Benefit of Dynamic Provisioning

We now show the benefit of dynamic provisioning (Section V-C). 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 9 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.*

VII. Conclusion 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. 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.

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

[5] 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.

[6] 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*, 2003, pp. 15–28.

[7] 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.

[8] 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.

[9] 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)*, 2004, pp. 119–130.

[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] W. Jian, W. Kwame-Lante, and G. Kartik, "Xenloop: a transparent high performance inter-vm network loopback," in *Proceedings of the 17th international symposium on High performance distributed computing*, ser. HPDC '08, 2008.

[13] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration," in *In Proceedings of the 39th International Conference on Parallel Processing*, 2010.

[14] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-allocation," in *In Proceedings of the International Workshop on Quality of Service*, 1999.

[15] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.

[16] R. Geambasu, S. D. Gribble, and H. M. Levy, "Cloudviews: Communal data sharing in public clouds," in *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

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