A Fine-Grained Adaptive Middleware Framework for Parallel Mobile Hybrid Cloud Applications

Reza Shiftehfar
Department of Computer Science
U. of Illinois at Urbana-Champaign
Email: sshifte2@illinois.edu

Kirill Mechitov
Department of Computer Science
U. of Illinois at Urbana-Champaign
Email: mechitov@illinois.edu

Gul Agha
Department of Computer Science
U. of Illinois at Urbana-Champaign
Email: agha@illinois.edu

***Abstract*— Mobile Cloud Computing (MCC) overcomes mobile device limitations by delegating tasks to more capable cloud spaces. Existing mobile offloading solutions generally rely on full virtual machine migration, which is coarse-grained and costly, or implementation of code offloading as part of the application logic, which greatly increases the application complexity and the associated software development costs. Some recent solutions implement fine-grained offloading, but pause the local mobile application while waiting for the offloaded code results. This leads to sequential execution and wastes local mobile resources and ignores the potential elasticity of the cloud environment. We have developed the IMCM framework to support parallel mobile application offloading to multiple cloud spaces. IMCM is fine-grained, supporting application distribution at the granularity of individual components; it is adaptive, addressing the dynamicity in run-time conditions and end-user contexts; and it is fully parallel, supporting both parallel application model and simultaneous execution at mobile device and multiple private and public cloud spaces. Our evaluation results show that IMCM can improve the performance of computationally intensive mobile applications by a factor of over 50, while masking the underlying complexity of mobile-to-cloud code offloading.**

I. Introduction

Mobile devices have become ubiquitous, but they are still constrained by their limited resources. Compared to laptops and desktops, mobile devices typically have weaker hardware, more restricted network access, and more limited access to energy. These limitations have created an increasing gap between the demand for more complex applications and the availability of required hardware resources [11]. Cloud computing has the potential to provide a solution to overcome mobile device constraints and to address the ever-increasing complexity of modern mobile applications. Cloud computing provides elastic on-demand access to virtually unlimited resources at an affordable price. The elastic resources allow weaker devices run more demanding applications by outsourcing storage or computation needs to cloud spaces. To achieve this, certain parts of mobile application have to be selected, sent to a remote cloud space, executed, and the results brought back to the mobile device. This process is known as *code offloading* and has been widely studied within the context of distributed systems and grid computing [2], [15], [19].

Current practical solutions for providing the code offloading capability for mobile-cloud applications rely on either hard-coding the offloading decisions as part of the developed program or using full Virtual Machine (VM) migration to make an exact copy of the running application within cloud space. The former has the advantage of being fine-grained, well-tuned, and potentially self-adapting based on run-time parameters, but it requires programmers to rewrite their mobile application in an offloadable format. This places a significant burden on application developers, requires structural changes for existing applications, and requires continuous maintenance, as mobile applications evolve over time. On the other hand, the latter approach is based on the assumption that running the same code on a faster machine improves the application performance. It has the advantage of not creating additional work for developers but is highly coarse-grained. Virtual machines are large components and moving them around is very expensive even within a local area network (LAN) [9].

Application component distribution between mobile device and cloud resources must be flexible to satisfy different user expectations, and adaptive to address dynamic run-time context changes. This requires *open systems* that interact with the environment while addressing application constraints, user expectations, and hardware limitations. Moreover, despite some theoretical support for opportunistic parallelism, most of the existing code offloading solutions pause local mobile execution while waiting for offloaded code result leading to semi-sequential applications [4], [6], [10]. With modern mobile devices benefiting from fast powerful multi-core processors, new offloading solutions are required that supports fully-parallel applications.

These considerations motivate us to develop a fine-grained adaptive solution that minimizes the required manual changes to applications, prevents creating additional work for programmers, and allows fine-grained adaptive distribution of application components. Our overall goal is to bridge the gap between mobile application development, cloud computing and dynamic adaptive code offloading while satisfying both application and end-user requirements. Our main design objective is to help mobile-cloud application programmers to focus on developing their application logic without worrying about component distribution, that would be performed transparently and dynamically at run-time. We propose a framework that masks all the complexity of the mobile application code offloading to multiple cloud spaces. We model mobile-cloud application as a composition of self-contained autonomous actor components. Our framework is fine-grained, supporting application configuration and distribution at the granularity of individual components, adaptive, addressing the dynamicity in environmental conditions and end-user contexts, and transparent, masking the underlying complexity of mobile-to-cloud code offloading. It supports component distribution in a hybrid cloud environment consisting of one or several public and private cloud spaces. Finally and most importantly, it provides a new code offloading model that supports parallel program execution where application components located at mobile device and different cloud spaces are executed independently but concurrently.

This paper makes the following contributions:

- We propose a new mobile code offloading model that supports fully-parallel program execution, and present a proof-of-concept implementation: the Illinois Mobile-Cloud computing Manager (IMCM).
- We highlight the advantages of the parallel execution of mobile applications on cloud spaces and demonstrate the value of simultaneous local mobile and remote cloud application execution in this framework.
- We study the impact of various run-time parameters on the effectiveness of application offloading and devise an adaptive solution to dynamically manage component distribution.
- Empirical evaluation results for a suite of benchmark mobile applications show a speedup factor between 9 and 56 times over sequential execution on a mobile device.

II. Related Work

Code offloading is not a new idea and has been used widely in grid computing, where processes are migrated within the same computing environment for the purpose of load balancing between different machines [2], [15], [19]. However, modern offloading era started when virtualization became popular allowing cloud vendors to run arbitrary applications from different customers. In recent years, with the popularity of mobile devices and the availability of affordable public cloud resources, code offloading has been extended to mobile devices and Mobile-Cloud Computing is introduced to overcome mobile limitations [11], [16]. Systems benefiting from MCC usually use one of the following two approaches: rely on the programmers to manually partition the program and specify how to offload parts of an application to remote servers, or to use full virtual machine migration in which entire process or entire OS is migrated to cloud space [17]. Former requires significant manual work and latter is too expensive [9].

In order to overcome manual work and expensive data transfer, both automatic partitioning solutions [8], [13], [14] and fine-grained code offloading solutions are required. MAUI [6] combines both and enables fine-grained energy-aware offloading of mobile code to a remote server. It uses a combination of virtual machine migration and automatic code partitioning. However, it only supports sequential execution where mobile device is paused while waiting for the offloaded code result. It also supports only single remote server and requires manual annotation of methods by programmer and offline static analysis of the source code before execution.

CloneCloud [4] avoids manual work and enables unmodified mobile applications to be offloaded. It supports offloading multiple methods at the same time but requires an exact clone of the mobile device on the cloud. Despite its theoretical support for opportunistic parallelism, it leads to sequential execution in practice, as the phone execution will be paused whenever local code accesses the migrated state. Its application partitioner is also static and needs to pre-process the application code in an offline mode. It considers limited input/environmental conditions in the offline-preprocessing and needs to be bootstrapped for every new application built.

ThinkAir [10] provides mobile code offloading while allowing on-demand VM creation and resource allocation. However, all VM creation is behind a single remote server and masked from the mobile device. In fact, its main focus is on VM load-balancing rather than mobile-cloud application offloading. It supports opportunistic parallelism that results in limited practical parallelism.

COS [9] combines VM migration with application-level migration to reach fine-grained load balancing. However, it is focused on load-balancing of VMs within a cloud space in order to improve overall cloud performance and does not consider the performance of individual applications.

Our framework is focused on improving individual application performance while addressing dynamic run-time environment, end-user context, and application behavior. Unlike previous research, our system supports offloading to multiple remote locations, concurrent application model, and simultaneous execution on both mobile device and remote cloud resources.

III. Cloud application and infrastructure model

In order to formulate application component offloading problem, a comprehensive mobile-hybrid-cloud application model is needed. This section summarizes our view on cloud, cloud-application, and mobile-cloud application.

A. Cloud Model

Over time, cloud services have moved from the model of using public cloud spaces to private clouds and recently to the hybrid model combining both [20]. Cloud infrastructure is traditionally provided by large organizations, thus referred to as public cloud. However, storing data on third-party machines suffers from potential lack of control and transparency in addition to legal implications [3]. To address this, cryptographic methods are used to encrypt the data stored in public cloud while decryption keys are only disclosed to authorized-users. However, these solutions inevitably introduce heavy computational overhead for continuous encryption and decryption of data, distribution of decryption keys to authorized users, and management of data when fine-grained data access control is desired [18]. Cryptographic methods do not scale well, have significant overhead, are expensive to maintain, and are often slow especially in widely geographically distributed environments such as cloud. Moreover, they have traditional data-centric view on the cloud limited to storing data and providing services for accessing it.

In modern mobile-cloud applications, resources stored in the cloud contain more than just data. These resources contain part of the application code that results in access operation meaning execution of the code inside the cloud. Certificate-based authorization systems fail to address this type of applications, as the encrypted piece of code within the cloud cannot be executed without decryption and revealing the content to the cloud provider. As a result, companies gradually moved toward building their own private clouds [3]. However, owning private datacenter is not as efficient, reliable, nor scalable as using the public ones. Thus, in recent years, a combination of both private and public cloud spaces is used that benefits from all the advantages of the public cloud while keeping the confidential or sensitive data and algorithms in-house [7]. Unlike previous mobile-cloud solutions that considers only one single remote location for offloading [4], [6], [10], our model considers a hybrid cloud space consisting of one or several private and public cloud spaces and allows concurrent application component offloading and execution on all of them.

B. *Cloud Application Model*

In order to replace the traditional data-centric view of the cloud with a more general data/computation-centric view, current popular service-oriented architecture [12], that provides services on data stored in the cloud to external users, needs to be replaced with a new architecture that dynamically and transparently leverage cloud resources to address end-user mobile device limitations. An elastic application development environment allows components storing data or performing computations to be transparently distributed between private clouds, public clouds, and end-user device. When such an application is launched, an elasticity manager monitors the environment, resource requirements of different application components, and makes decision about component distribution between mobile device and different cloud spaces based on run-time parameters, application behavior, and user expectations. This allows mobile applications to adapt to different workloads, performance goals, energy limitations, and network latencies. In order to prevent creation of additional work for application developers, unnecessary details of distribution and move-around of application components should be masked.

In order to reach the maximum level of parallelism without the hassle of traditional multi-threading model, modern cloud-based applications avoid using shared memory model that is unnatural for developers and leads to error-prone non-scalable programs [12]. Instead of relying on global variables and shared states, modern cloud-based applications restrict the interaction between various components to communication using messages. This approach to cloud application development aligns with the concepts of *actor model of computation* [1] that sees distributed components, called actors, as autonomous objects operating concurrently and asynchronously. In response to a received message, an actor can make local decisions, create new actors, send more messages, or change its behavior to respond differently to the next received message [1]. Compared to the traditional shared memory model, actors are a better fit for highly dynamic applications operating in open and challenging environments. Actors may be created and destroyed dynamically, they can change their behaviors, and migrate to different physical locations. The model provides *natural concurrency*, *resiliency*, *elasticity*, *decentralization*, *extensibility*, *location transparency*, and *transparent migration* that ease the process of scaling-up or out, which is a critical requirement for cloud-based applications. As a result, our view of a mobile-cloud application consists of actors distributed between local mobile device and different cloud spaces.

IV. Approach

Mobile-cloud computing relies on code offloading process to benefit from available remote cloud servers. Running applications in VM and migrating the entire VM to a more resourceful machine allows benefiting from offloading without processes even knowing of the migration. However, VMs are usually large in size and migrating them is costly even when performed within a local area network [9]. An alternative that prevents coarse-grained data transfer is to migrate application components. As a result, offloading process consists of decision making about appropriate parts to offload in addition to migrating them, executing them on remote servers and bringing back the results. Proposed actor-based mobile-cloud application model provides natural application partitioning and masks component migration process. The only remaining piece is finding appropriate components for offloading and this section focuses on making such optimal offloading decision with respect to target goal, application behavior, and run-time parameters.

A. *Offloading Decision for Sequential Application to Single Remote Server*

Without considering offloading process cost and its effect on application behavior, speedup resulting from running the same code on a more resourceful machine can be defined as the ratio of available resource:

$$Speedup = \frac{S_s}{S_m} = \frac{F_{server} * C_{server} * X_{server}}{F_{mobile} * C_{mobile}} \tag{1}$$

where S_s, F_{server}, C_{server}, S_m, F_{mobile}, C_{mobile} are the speed, processor frequency and number of cores of the server and mobile device. X_{server} is the additional speedup resulting from availability of additional resources on the remote server, e.g. caches, memory and potentially more aggressive pipelining. Equation 1 states that offloading is always beneficial, as long as there is a more resourceful server. However, it ignores the required resources for the offloading process and the effect of offloading on application behavior. However, only if the required amount of resources for offloading process is small, network connection is fast, and amount of transferred data is small, speedup close to Equation 1 can be achieved in practice.

For most applications, the required resources for offloading process cannot be ignored.

Extending Equation 1 to include the cost of offloading highly depends on the target offloading goal. Offloading goals can vary significantly based on the application or user and range from maximizing the application performance (e.g. games, vision-based applications) to minimizing energy consumption on the mobile device (e.g. background applications). This paper focuses on *maximizing application performance* goal and leaves the remaining goals to future effort.

Maximizing application performance, or minimizing total execution time, provides real-time applications with higher quality computation in the same amount of time leading to a smoother and better experience for users. Assuming a small application with w amount of offloadable work, the goal is to decide whether to offload or not. Following [11] model, we can summarize the problem as below:

$$\frac{w}{S_m} > \frac{d_i}{B} + \frac{w}{S_s} \rightarrow w * (\frac{1}{S_m} - \frac{1}{S_s}) > \frac{d_i}{B} \tag{2}$$

where S_m and S_s are the speed of the mobile device and remote server processors, B is the network connection bandwidth and d_i is the size of data to be transferred. The left side of this equation shows the total required time to execute work w on the mobile device while the right side captures the required time to transfer data to a remote server and execute it on the server. It only makes sense to offload when the left side is larger than the right side. Note that this equation ignores many parameters such as communication latency, required time to bring back the result, etc. Equation 2 also shows that S_s effect is of second degree and an infinitely fast server ($S_s = \infty$), does not lead to an always-offloading decision, if other parameters are not proportional.

Although we focused on the goal of maximizing application performance, offloading decision for the goal of minimizing mobile device energy consumption is similar for *sequential* applications. In sequential execution, mobile device remains in idle state consuming energy while waiting for the results from the offloaded code. Consequently, the required time for application execution on the remote server is proportional to mobile device energy usage [4], [6], [10]. However, this effect is limited to sequential applications where only one of the mobile device or remote server executes code at any time.

Use of Equation 2 leads to a pause-offload-resume model where the system pauses before executing any part, checks Equations 2 and decides whether to offload or not. If decision is to offload, mobile application will be paused, data transferred to remote server, code executed on remote server, results brought back to the mobile device, and mobile application resumed [6]. However, in communication-intensive applications, offloading single components at a time results in significant remote communication. When components are on the same device, communications are relatively fast and through shared memory space. But when placed on different machines, communications go through multiple network devices and become costly. As a result, components communicating extensively should be offloaded together. The problem of deciding on offloading multiple parts of an application can be formulated as a graph partitioning problem where nodes are application components, having a weight equal to the amount of their computation, and edges are communications in between, having a weight equal to the amount of transferred data. In such a graph, offloading decision equals finding the minimum cost cut to partition the graph between mobile device and remote server [4], [10]. Note that application execution is still considered sequential and only one of the components will be executed at any time.

In order to avoid sequential program execution resulting from previous graph-based partitioning approach, CloneCloud [4] and ThinkAir [10] support opportunistic parallelism. When a component is offloaded, the remaining code on the mobile device continues with its execution as long as the offloaded state is not accessed. As soon as the local code tries to access the state of the offloaded part, local execution is blocked and only resumed when the offloaded code result is received. Despite theoretical potential for parallel execution, this model still leads to sequential execution in practice. In most applications, shared program state is constantly accessed by different parts and mobile code execution remains blocked most of the time. Moreover, it can only considers single remote location for offloading. This is one of the main drawbacks of using a shared state program model and a natural result of sequential applications. When parallelism is considered, mobile device and remote server can execute code simultaneously in addition to multiple remote servers working concurrently.

B. Offloading Decision for Parallel Applications to Hybrid Cloud Environment

Deciding on optimized offloading plan for parallel applications in a hybrid cloud environment requires considering application type, available resources at different remote machines, and offloading effect on future application behavior. Similar to previous sections, target offloading goal is maximizing application performance or minimizing total application execution time. We still have a graph G(V,E) where vertices represent application components and edges represent communications in between. The goal is to partition the graph between mobile and different cloud resources in a way that total execution time is minimized. Total execution time consists of the time required to execute the application code in addition to the time required for remote components to communicate and exchange data with each other. Fully parallel execution refers to both parallel execution on multiple remote locations and simultaneous local and remote execution. In other words, mobile device and different cloud spaces execute their components simultaneously. As a result, total application execution time is the maximum time required for any of the mobile or remote spaces to finish executing program code for all of its assigned components. Since local communication between components located on the same machine is relatively fast, we can ignore local communication and only consider communications between different components placed at different locations.

Note that different locations can communicate simultaneously and the total required time for communication is equal to the maximum communication time of different locations. Using table I notations, the offloading goal can be summarized as following:

TABLE I: Notations used in parallel offloading model

Notation	Description
$B(L)$	Connection bandwidth out of location L
CommAtLoc(L)	Communication time from components on Location L to all other locations
Cores(L)	Number of cores available at Location L
Δ	Time interval of running elasticity manager
Exec(i,l)	Exec. time of component $i \epsilon [1, N]$ at location $l \epsilon [0, M]$
ExecAtLoc(L)	Execution time for all components on Location L
JobCount(i)	Number of requests processed by component i during the time interval Δ
Loc(i,t)	Location of component i at time t
$LocAllowed(i, t)$	Set of locations at which component i is allowed to be placed at time t. $LocAllowed(i, t)\ \epsilon\ [0, M]$
$LocEQ(L_1, L_2)$	Checks whether two given locations are identical. Returns 1, if $L_1 = L_2$. Otherwise, returns 0.
MaxAppPerf	Maximum Application Performance
MinAppExec	Minimum Application Execution Time
$ProfComm(i, j)$	Profiled amount of communication between components i and j during the time interval Δ

$$Max(\ MaxAppPerf\ ) = Min(\ MinAppExec\ ) = \\ Min(\ \max_{0 \leq L \leq M}(ExecAtLoc(L))\ +\ \max_{0 \leq L \leq M}(CommAtLoc(L))\ ) \tag{3}$$

Mobile application consists of N components and each component $i \epsilon [1, N]$ is located at $Loc(i, t)$ at time t. Having M different cloud spaces results in $Loc(i, t) \epsilon [0, M]$ where 0 represents local mobile device and $[1, m]$ corresponds to different cloud spaces. Assuming that we know the application component distribution between the local mobile device and the hybrid cloud spaces at time t_1, our goal is to find optimal component distribution for next time interval t_2 in a way that application performance is maximized. Thus, different parts of Equation 3 can be extended as following:

$$\max_{0 \leq L \leq M}(\ ExecAtLoc(L))\ = \\ \max_{0 \leq L \leq M}(\ \frac{1}{Cores(L)} * \sum_{i=1}^{N}\{LocEQ(L, Loc(i, t_2))* \\ Exec(i, Loc(i, t_2)) * JobCount(i)\}\ ) \tag{4}$$

Note that both $Exec(i, L)$ and $JobCount(i))$ are provided by the monitoring system and are results of previous profiling of the application. $LocEQ(L_1, L_2)$ considers the execution time of only components running on location L. Similarly, the second part of Equation 3 can be extended as below:

$$\max_{0 \leq L \leq M}(\ CommAtLoc(L))\ = \\ \max_{0 \leq L \leq M}(\ \frac{1}{B(L)} * \sum_{i=1}^{N}\sum_{j=1}^{N}\{LocEQ(L, Loc(i, t_2))* \\ (1 - LocEQ(L, Loc(j, t_2))) * ProfComm(i, j)\}\ ) \tag{5}$$

As mentioned before, this equation shows the maximum required time for one of the locations to send out all its communications. $LocEQ(L, Loc(i, t_2))$ considers only components that will be located at Location L at time t_2 and $(1 - LocEQ(L, Loc(j, t_2)))$ captures only remote communications out of location L. Solving these equations results in a set of $Loc(i, t_2)$ that are the optimized locations for different application components during the next time interval Δ. But not all components of an application are offloadable. So, a few constraints must be added to the above optimization problem. As we are considering a hybrid cloud consisting of multiple private and public cloud spaces, application developers or users can specify additional constraints in terms of how different components can be offloaded to different locations. These additional constraints can address certain privacy issues in terms of not offloading sensitive or confidential components to public cloud spaces. Required constraints can be written as below:

subject to constraints:

$$Loc(i, t_1)\quad \epsilon \quad LocAllowed(i, t_1)\ :\ \forall\ \ i\ \epsilon\ [1, N] \\ Loc(i, t_2)\quad \epsilon \quad LocAllowed(i, t_2)\ :\ \forall\ \ i\ \epsilon\ [1, N] \\ \sum_{i=1}^{N}\sum_{j=1}^{N}\{LocEQ(L, Loc(i, t_2)) * (1 - LocEQ(L, Loc(j, t_1)))\} \\ \leq\quad \alpha * Cores(L)\ :\ \forall\ \ L\ \epsilon\ [0, M] \tag{6}$$

The last constraint is added to prevent flooding too many components at once to a remote server with good initial performance. We limit the number of components that can be offloaded to each remote server to a factor of the number of available cores on that server. α of range 2 to 8 is compatible with our evaluation results that shows best performance can be achieved when 2 to 8 actors are assigned to each core. If after one round of component move around the target remote server still has enough resources and the execution times are still fast enough, another round of actors can be migrated to that location. In most cases, $LocAllowed(i, t_1) = LocAllowed(i, t_2)$, as the privacy constrained are not often changed. However, the user or the run-time environment has the option of adjusting privacy requirements at run-time whenever needed.

V. Evaluation

This section presents our experimental results for evaluating our proposed framework. To make the results comparable and link them to our target offloading goal of maximizing application performance, we measure effectiveness as the speedup gained compared to sequential local execution on mobile device. Our selected corpus consists of applications covering different types of programs: CPU intensive, communication intensive, I/O intensive, and combined. In subsection V-C, we investigate the effect of different application parameters on offloading decision. In subsection V-D we evaluate the performance of the proposed middleware framework.

A. Experimental Setup

Our used equipment include a Samsung Google Nexus S as the mobile device and a Macbook Pro Laptop as the remote offloading server. Table II summarizes the specifications of our used equipment. Mobile device and the remote server are both on the same WiFi network.

TABLE II: Specifications of the used equipment for evaluation

	Remote Server	**Mobile Device**
System	Macbook Pro-Retina	Samsung Google Nexus S
OS	Mac OSX 10.9.4	Android 4.1.2
VM	JVM (JRE 1.6)	DalvikVM
Processor	Intel Core i7	ARM Coretex-A8
Proc. speed	2.3 GHz	1 GHz
No. of cores	4	1
L2 Cache	256 KB/Core	256 KB
L3 Cache	6MB	-
Memory	16 GB	512 MB

Our mobile-cloud application model is based on the actor model of computing that offers natural parallelism for developed applications. Many actor programming languages have been developed over years to support different applications. Despite some small differences, most of these programming languages provide main standard actor semantics including *encapsulation*, *fair scheduling*, *location transparency*, *locality of references*, and *transparent migration*. For our experiments, we chose Salsa ([21]) as the programming language mainly due to its adherence to standard actor semantics. Salsa provides good support for parallel and distributed programming. Its support for code and data mobility and asynchronous message passing makes programming for distributed systems a natural task. Its coordination model provides an attractive feature for parallel programming where multiple CPUs need to coordinate and communicate between themselves in an efficient manner. SALSA depends on Java, hence it inherits Java's powerful feature of portability across different platforms [21]. We were able to make SALSA work on Android mobile devices running DalvikVM with some modifications. Salsa provides lightweight actors. The use of lightweight actors makes SALSA highly scalable that is one of the main limitations of some older actor languages. A huge advantage of using lightweight actors is the speed and ease of actor migration between different devices. Our experimental result showed that Salsa actors can be created or migrated in $100 \sim 200$ ms on or between different machines working on the same WiFi. This fast migration speed eases the process of mobile-cloud application offloading.

The base case in our evaluation is the required time for local sequential execution of the application on the mobile device and the execution speedups are used for comparing different scenarios. In order to account for randomness, we repeat each experiment five times and verify the statistical significance of observed execution times through non-parametric Mann-Whitney U-tests. Unless stated otherwise, the test is two-tailed and the significance level is $\alpha = 0.01$.

B. Program Corpus

Table II lists the programs used in the evaluation together with their main characteristics. Evaluation benchmark programs are selected based on their characteristics to cover different application behavior: Computational intensive, Communication intensive, and I/O intensive. In addition, a multi-behavior application is added to combine different characteristics. To avoid a bias towards specific strengths of our approach and to foster comparability, we mostly use similar examples as for works presenting solutions to mobile-cloud computation offloading. The *NQueen* program is a computation-intensive application that places N queens on a $N * N$ chessboard so that no two queens threaten each other [10]. The *Heat* program is a communication-intensive application that simulates heat transfer in a two-dimensional grid in an iterative fashion [9]. Our implementation allows specifying the desired level of communication and both medium and high level of communications are studied. The *Trap* program is a computation-intensive application that calculates a definite integral by approximating the region under the graph as a trapezoid and calculating its area. The *Virus* program reads in file streams from disk and scans for the signature of a given virus [4], [10]. The *Rotate* program is an I/O-intensive application that reads in an image from disk, rotates it in memory and writes it back to disk. Similarly, the *ExSort* program is an I/O-intensive application that sorts the content of a large file using external sort algorithm in limited amount of memory. Finally, the *Image* program combines all I/O, CPU, and communication characteristics by detecting and recognizing all faces in a given picture using a large dataset of known faces [4]–[6], [10]. Since processing of each picture is performed sequentially, multiple images are processed simultaneously in order to add parallelism. To save space, only part of the results are presented in this paper.

C. Influence of App. Parameters on Offloading Decision

This section discusses how different application or execution properties influence offloading decision, answering the following research questions: (RQ1-RQ3)

RQ1: What influence do the a) cost of offloading process, b) application type, and c) run-time parameters have on the mobile-cloud offloading decision?

Table III shows the speedup results for different applications together with applications' main characteristics. While *raw speedup* column ignores the cost of offloading process, *offload speedup* column shows a more realistic view on mobile-cloud offloading by including the required time for offloading process. Note that different rows of the table represents different applications with significantly different behavior, architecture and characteristics that should not be compared with each other. Comparing the values of *raw speedup* and *offload speedup* columns shows the effect of offloading cost on gained speedup. Offloading cost includes the required resources to make offloading decision, offload the application code to remote server and bringing back the result. Ignoring the cost of offloading process, Equation 1 predicts the speedup resulting

TABLE III: Programs used to evaluate our framework. Application characteristic column shows dominant behavior of the application, raw speedup column summarizes maximum speedup gained by running application on a more-resourceful machine excluding offloading time, and offload speedup shows maximum speedup resulting from offloading including offloading overhead

Experiment	Description	Application Characteristic				Raw Speedup	Offload Speedup
		Comp.	Comm.	I/O			
				read	write		
NQueen	Places N Queens on N*N board	intensive	-	-	-	73	56
Image	Detects & recognizes all faces in a photo	intensive	limited	limited	-	91	44
Trap	Uses trapezoidal rule to calculate definite integral	intensive	limited	-	-	30	21
Virus	Scans a file stream for a specific virus signature	-	-	intensive	-	28	21
Rotate	Reads, rotates & saves an image to disk	-	-	intensive	intensive	28	9
ExSort	External Sort of the content of a file	intensive	-	intensive	intensive	46	36
Heat1	simulates heat exchange on a board	limited	medium	-	-	31	29
Heat2	simulates heat exchange on a board	limited	high	-	-	14	14

from running the same code on a faster machine. Assuming $X_{server} = 7$ for our experimental setup, the expected speedup is as below:

$$Speedup = \frac{S_s}{S_m} = \frac{2.3 * 4 * 7}{1.0 * 1} = 64 \quad (7)$$

raw speedup column shows that a speedup of 64 times or even higher is possible. However, when large amount of data needs to be offloaded (such as *Rotate* application), offloading speedup reached in practice is significantly lower. Moreover, the result highly depends on the application type and behavior as well. A computational-intensive application with high degree of parallelism (e.g. *NQueen*) can benefit from all the additional available resources on the remote server and can reach a high offloading speedup. Extensive I/O operations or communications between different components limits application's ability of benefiting from additional available computational resources at the remote server and reduces the gained speedup (e.g. *Rotate* and *Heat*).

In order to decide on the beneficiary of offloading w amount of computation to a remote server for our experimental setup, Equation 2 can be used with values from table II:

$$w * (\frac{1}{1024Mhz} - \frac{1}{2.3 * 1024MHz * 4 * 7}) > \frac{d_i}{B} \quad (8)$$

Rearranging the equation results in $B_{min} \geq 1040 * \frac{d_i}{w}$ to be the minimum required bandwidth in order for offloading decision to reduce total application execution time. The equation depends on the ratio of $\frac{d_i}{w}$ and can only be true when the ratio is small enough. In other words, application offloading is beneficial for large amount of computation (w) and low amount of transferred data (d_i). For values in between, the decision depends on the available bandwidth (B) and an elasticity manager must evaluate the equation based on run-time parameters. For *N-Queen* problem, a single integer value has to be transferred both for input value (N) and final result and d_i is very small. At the same time, problem is computational-intensive and requires large amount of computation (large w). As a result, any type of network connection provides enough bandwidth and offloading always improves application performance. Note that the code of the N-Queen solver is assumed to be available on the remote server and network latency is ignored. In case of the *Image*, assuming remote server to be super fast ($S_s = \infty$), offloading decision depends on w, d_i and B. If detection of faces in the initial image, extracting features for every detected face and comparison to database are all offloaded, the entire initial image needs to be transferred to the remote server and the amount of communicated data (d_i) is large. Thus, it is only beneficial to offload, if B is large enough. On the other hand, if the initial detection of faces are performed locally and only the extracted features are transferred, d_i is much smaller. Consequently, even for slower network connections, offloading of the remaining parts is beneficial. This highlights the importance of considering the combination of all parameters for deciding on offloading. Different parts of an application can become offloading candidates at different time and an elasticity manager is required to dynamically decide on offloading based on run-time parameters.

RQ2: How significant is the influence of problem size (amount of work) on mobile-cloud offloading?

Figure 1 and Figure 2 show the offloading speedup for different amount of work for *NQueen* and *Image* applications. The results show that larger amount of work results in more computationally-intensive applications, reduces the importance of the fixed amount of work required for offloading process, and increases the gained speedup. While initial offloading speedup of *NQueen* problem is almost equal to 1 (for N=8) due to low amount of required computation, changing N value exponentially increases the amount of work to be performed and the resulting speedup. *Image* problem is a multi-behavior application with initial speedup of larger than 1 due to the size of computations required for processing even one single image. For this problem, changing the amount of work equals increasing the number of images to be processed and results in linear increase of speedup.

RQ3: What influence does the application parallelism degree have on mobile-cloud offloading?

Equation 7 predicts the ideal speedup resulting from offloading where computation is large enough, code has high degree of parallelism roughly comparable to available resources, and negligible amount of resources is used for offloading process. Without benefiting from parallelism, running the same code on a more resourceful machine can only provide limited speedup (Sequential remote execution graphs of Figure 1 and Figure 2). This speedup is mostly because of benefiting from remote server's faster CPU speed, additional available caches, and

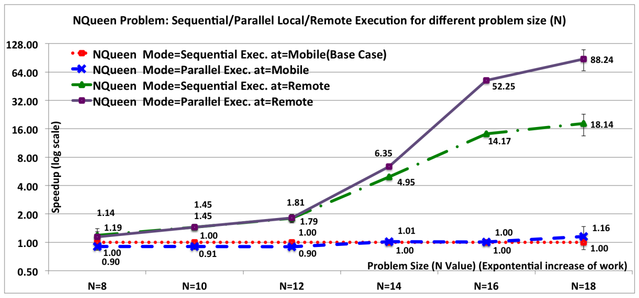

Fig. 1: Speedup summary for local and remote execution of N-Queen execution for different amount of work (different problem size)

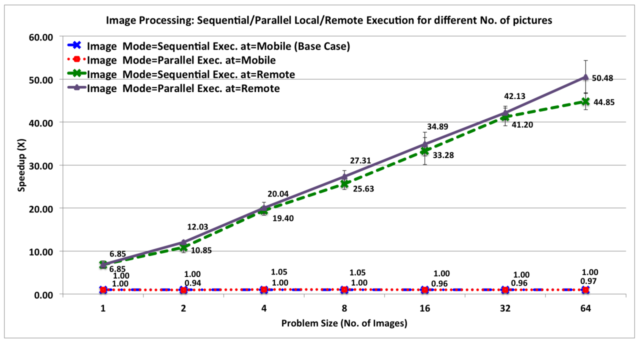

Fig. 2: Speedup summary for local and remote execution of Image Processing application for different amount of work (different no. of images to process)

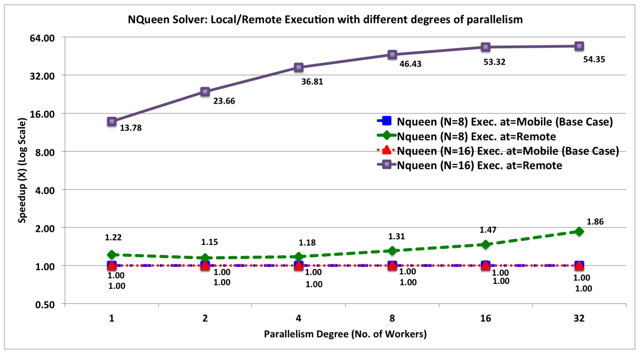

Fig. 3: Speedup summary for local and remote execution of NQueen problem with different degree of parallelism

more memory. However, additional available processing units are not used. We mentioned that for practical applications, the amount of resources required for offloading process is negligible compared to resources required for performing large amount of computation. If computation is not large enough, even using high degree of parallelism does not provide significant additional speedup. However, when the amount of computation is large enough, higher degree of parallelism significantly improves the performance and the benefit of having additional processing resources becomes visible.

Figure 3 shows the relationship between application parallelism degree and speedup resulting from offloading. While on a mobile device with only one single core increasing parallelism degree does not improve the performance, on a more resourceful remote server increasing the program parallelism degree allows better utilization of resources and increases application performance. While sequential execution of *NQueen* problem on a faster system generates a speedup of 14 times, increasing the parallelism degree increases the resulting speedup to 55. Performance improvement resulting from increasing program parallelism degree is limited by the availability of resources. At a certain parallelism degree, resources will become saturated and further increase of parallelism degree will have reverse negative effect (Figure 5). Considering the null hypothesis that remote sequential execution is as effective as the remote parallel execution, Mann-Whitney U-test shows that all differences for various problem sizes and parallelism degrees are significant. Consequently, the null hypothesis is rejected.

D. *Effectiveness of the Proposed Middleware Framework*

This section discusses the performance of the proposed IMCM middleware framework, answering the following research questions: (RQ4-RQ6)

RQ4: Is the IMCM proposed parallel local & remote execution offloading solution more effective than existing sequential (or pseudo-parallel) execution offloading solutions?

While offloading computation to a more resourceful system can improve overall application performance, mobile device local resources are wasted while waiting in the idle state for the result of offloaded code to be returned. With mobile devices becoming more powerful, this wasted computational power can be put to a better use. Our proposed framework supports simultaneous local and remote application execution and uses local mobile resources to execute other parts of an application while waiting for the offloaded code result. Figure 4 shows the speedup differences between processing different number of images using only remote server and simultaneous execution on both local device and remote server. Since processing of a single image is sequential, for small amount of work (small number of pictures to process), total execution time will be dominated by the required time for local mobile device to process its share. This will result in remote server starvation and waste of resources, as there will be no more job for it to process. However, with increase in the amount of work, there will always be enough job for remote server to perform and the advantage of using both local and remote server for application code execution becomes visible. Figure 5 shows the same effect based on application parallelism degree. We mentioned earlier that higher degree of parallelism will increase the flexibility of the application and results in higher offloading speedup. However, this is only true, if enough computational resources are available. As can be seen in the graph, increasing the parallelism degree (number of workers) initially results in higher speedup but after a certain point this effect is reversed. In fact, having higher degree of parallelism than the available resources results in over-saturation of resources, adds the overhead of managing

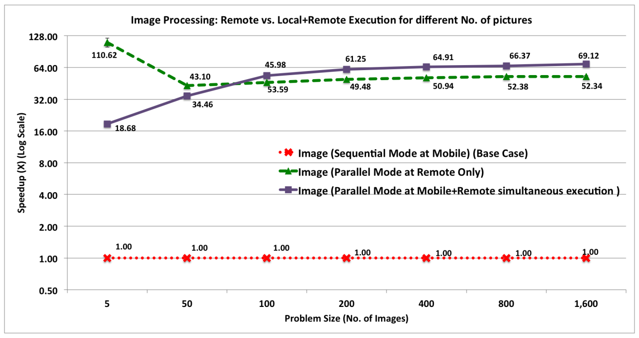

Fig. 4: Speedup summary for remote execution vs. local+remote execution of image processing problem with different problem size (different number of images)

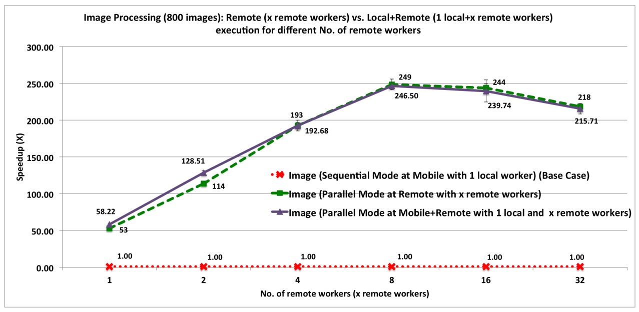

Fig. 5: Speedup summary for remote execution (x remote workers) vs. local+remote execution (1 local + x remote workers) of image processing problem with different number of remote workers

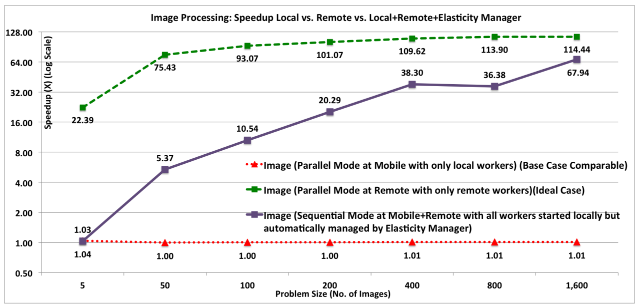

Fig. 6: Speedup summary for local execution (base case) vs. remote execution (ideal case) vs. local execution with elasticity manager (all automatic management) of image processing problem with different problem size (different number of images to process)

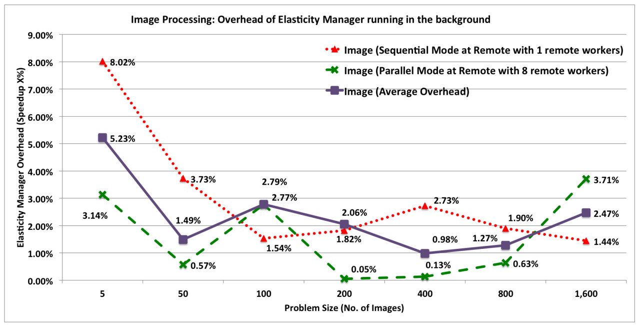

Fig. 7: Overhead resulting from elasticity manager for image processing problem with different problem size (different number of images to process)

all those workers, and reduces overall speedup. Our results show that required parallelism degree for an application to reach highest speedup is proportional to number of processing cores available. The coverage differences of any two different number of workers for both remote and simultaneous local and remote executions are significant ($\alpha = 0.01$). Thus, the null hypothesis that there is no significant difference between image processing execution with different number of workers can be rejected.

RQ5: How effective is the IMCM elasticity manager in detecting application run-time environmental parameters and offloading appropriate application components?

Despite significant performance speedup resulting from offloading application to more resourceful systems, manual configuration of components between local mobile device and remote server is not possible. Ideal component distribution depends on several factors that can dynamically change during execution. Thus, an elasticity manager is required to monitor environmental changes and find optimal offloading plan. Figure 6 shows the result for manual placement of application components versus automatic component management using IMCM elasticity manager that solves Equation 5 and Equation 6. Implemented elasticity manager uses the previous profiled execution times of different components at various locations to find the optimal location for placing every component for next interval. We currently do not use profiled execution time from previous execution of the application. As a result, there is an initial lag between start of an application and optimal placement of components resulting from the required time to collect enough profiled data. As a result, when problem size and resulting total application execution time increases, the gap between ideal placement of component and automatic distribution becomes narrower.

RQ6: What is the performance overhead of the IMCM automatic elasticity manager?

While offloading appropriate components to a remote server can potentially improve application performance, having a costly elasticity manager to profile run-time and application parameters and finding optimal distribution plan can result in less overall performance. Figure 7 shows the overhead results from our implemented elasticity manager. Results show that having profiler and elasticity manager running in the background generates $1-5\%$ speedup decrease on average. Considering the range of $9-60$x for speedup gain from offloading applications shows that IMCM elasticity manager overhead is insignificant. Moreover, as the problem size increases, the benefit of offloading becomes more dominant and the elasticity manager overhead becomes even less important.

VI. Limitations and Future Work

We try to ensure *conclusion validity* of our evaluation by checking the statistical significance of measured execution times with a robust non-parametric test at a high level $\alpha = 0.01$. One threat to the *construct validity* of our experiment is the use of performance speedup as effectiveness metric. With the amount of work increased, the gap between local mobile execution and other form of execution becomes larger. This reflects the improved performance and can be used to evaluate the performance of one application with different settings. However, the amount of work performed by different applications varies significantly. Moreover, different applications have different behavior, architecture and characteristics. Thus, comparison of speedup between different applications cannot be performed. The *external validity* of our evaluation is threatened by our focused corpus. Despite most of programs being selected according to benchmarks used in previous works, the corpus does not constitute a random sample of programs. Consequently, our results may generalize poorly. A larger study would mitigate this risk and is considered as future work.

Although we mentioned different component distribution plan in case of parallel execution for optimizing application performance and mobile energy consumption, this paper focuses only on mobile application performance improvement using code offloading. We are currently extending the framework to support mobile energy consumption optimization and to allow dynamic adjustment of application target goal. A big challenge with energy optimization is profiling detailed consumption of individual application components. While execution time of different components can individually be recorded using system clock, mobile device only reports lump sum energy consumption and break down of total energy among different components remains as a challenge.

VII. Conclusion

In this paper we proposed IMCM middleware framework for transparent automatic code offloading from mobile devices to hybrid cloud spaces. The framework is fine-grained, supporting application configuration and distribution at the granularity of individual components; it is adaptive, addressing the dynamicity in run-time conditions and end-user contexts. It further supports component distribution in a hybrid cloud environment consisting of one or more public and private cloud spaces. Finally, it provides a new code offloading model that supports parallel program execution where application components located at mobile device and different cloud spaces are executed independently but concurrently. Evaluation results show that the offloading result depends on application behavior, offloading cost, and run-time parameters and can range between 9 to 56 times.

VIII. Acknowledgments

The authors would like to thank members of the Open Systems Laboratory (OSL) group at University of Illinois at Urbana-Champaign for their valuable discussion and insight. This publication was made possible in part by sponsorship from the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084.

References

[1] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41. Springer, 1985.

[2] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 87–92. ACM, 2002.

[3] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90. ACM, 2009.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[5] B.-G. Chun and P. Maniatis. Dynamically partitioning applications between weak devices and clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, page 7. ACM, 2010.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[7] R. K. Grewal and P. K. Pateriya. A rule-based approach for effective resource provisioning in hybrid cloud environment. In *New Paradigms in Internet Computing*, pages 41–57. Springer, 2013.

[8] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI*, volume 99, pages 187–200, 1999.

[9] S. Imai, T. Chestna, and C. A. Varela. Elastic scalable cloud computing using application-level migration. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 91–98. IEEE Computer Society, 2012.

[10] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[11] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[12] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[13] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *ACM SIGPLAN Notices*, volume 40, pages 221–232. ACM, 2005.

[14] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI*, volume 9, pages 395–408, 2009.

[15] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. *Agile application-aware adaptation for mobility*, volume 31. ACM, 1997.

[16] M. Rahman, J. Gao, and W.-T. Tsai. Energy saving in mobile cloud computing*. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 285–291. IEEE, 2013.

[17] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[18] R. Shiftehfar, K. Mechitov, and G. Agha. Towards a flexible fine-grained access control system for modern cloud applications. In *Cloud Computing (CLOUD), 2014 7th IEEE International Conference on*, pages –. IEEE, 2014.

[19] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Software Architecture*, pages 29–43. Springer, 2002.

[20] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[21] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.