Parametric Analysis of Mobile Cloud Computing Frameworks using Simulation Modeling

Arani Bhattacharya[1], Ansuman Banerjee[2], and Pradipta De[1]

[1] SUNY Korea and Stony Brook University
{arani,pradipta.de}@sunykorea.ac.kr,
[2] Indian Statistical Institute
ansuman@isical.ac.in

Abstract. Mobile Cloud Computing (MCC) frameworks implement mechanisms for selecting tasks in an application and offloading those tasks for execution on a cloud server. Task partitioning and task offloading aim to optimize performance objectives, like lower energy usage on mobile devices, faster application execution, while operating even in unpredictable environments. Offloading decisions are influenced by several parameters, like varying degrees of application parallelism, variable network conditions, trade-off between energy saved and time to completion of an application, and even user-defined objectives. In order to investigate the impact of these variable parameters on offloading decision, we present a detailed model of the offloading problem incorporating these parameters. Implementations of offloading mechanisms in MCC frameworks often rely on only a few of the parameters to reduce system complexity. Using simulation, we analyze influence of the variable parameters on the offloading decision problem, and highlight the complex interactions among the parameters.

Keywords: Mobile Cloud, Simulation, Modeling, Resource Optimization, Integer Linear Programming, Application Offloading, Graph Partitioning

1 Introduction

Mobile Cloud Computing (MCC) presents the opportunity to utilize unlimited resources of cloud based infrastructure to augment resource constrained mobile devices. Prototype implementations of MCC frameworks have demonstrated that offloading computation can significantly reduce energy consumed to execute an application on a mobile device [1, 2]. The key principle in MCC is to profile the energy footprint of individual tasks in an application, and then utilize the information to offload execution of energy hungry tasks to a cloud server to optimize energy usage on the mobile device. Task partitioning and task offloading decisions are constrained by several factors, like communication energy to offload the program states to cloud, network latency affecting application completion time, and tasks, involving sensors, which must be executed natively on the device. There are implementations that trade-off among these constraints [3, 4]. However, practical operating environment of a MCC framework is more complex due to several variable parameters, like network conditions, runtime workload, and hardware characteristics.

The key challenge in designing practical MCC frameworks is to adapt to changes in the operating environment. Variations in network conditions is one of the hardest to cope with. It has been shown that dynamically adapting offloading decision based on varying network bandwidth can improve performance [5, 6]. Similarly, Zhang et al. showed the benefits of dynamically adapting data transmission rate to the cloud in presence of stochastic wireless channel errors [7]. Application workload is another source of variability to address while making offloading decisions. Exploiting dynamic execution patterns of an application can lead to better offloading decisions [8]. Barbera et al. implemented a tightly coupled device-cloud operating system that can overcome variations at different levels [9]. Even the diversity in smartphone hardware can lead to different offloading choices. For example, Lin et al. proposed the use of coprocessors, like GPUs in handheld devices, to arrive at better offloading decisions than those shown before [10]. The recurring theme in these works is that dynamic adaptation plays a crucial role in making better offloading decisions in MCC systems.

We observe that although system implementations have been effective in delivering performance gains, there is still a lack of in depth understanding of how individual parameters impact performance, as well as, influence each other. Given the complexity of these parameters, it is difficult to design controlled experiments in real environments. Therefore, we propose a simulation model that incorporates the parameters in a single model. This enables us to understand the interactions among different parameters that affect the offloading decision problem.

We summarize our contributions in this paper as follows:

- We propose a formal model that incorporates different parameters that influence the task partitioning and task offloading in MCC systems.
- We analyze how various parameters used in offloading decision affect the performance of MCC systems. We report how optimization objectives, viz. energy consumed on a mobile device, and application execution time, are affected by various parameters, like application and cloud server features, degree of parallelism exploited and network characteristics.

The rest of this paper is organized as follows. Section 2 discusses the working of an MCC offloading system. Section 3 explains the formulation. Section 4 shows the experiments and the corresponding inferences drawn. Section 5 concludes this paper.

2 System Model

In this section, we present the architecture of a mobile cloud computing (MCC) system. The models of different components of the system, such as mobile application, communication network and the cloud system, are based on this architecture.

Fig. 1 shows the architecture of an MCC system. An offloading decision engine partitions an application into two parts: one that executes on the mobile device, while the other is migrated to the cloud servers for execution. Communication between the mobile device and the cloud server uses the wireless interface on the mobile, which can be 3G, LTE or Wi-Fi enabled. We assume that the application source code resides on

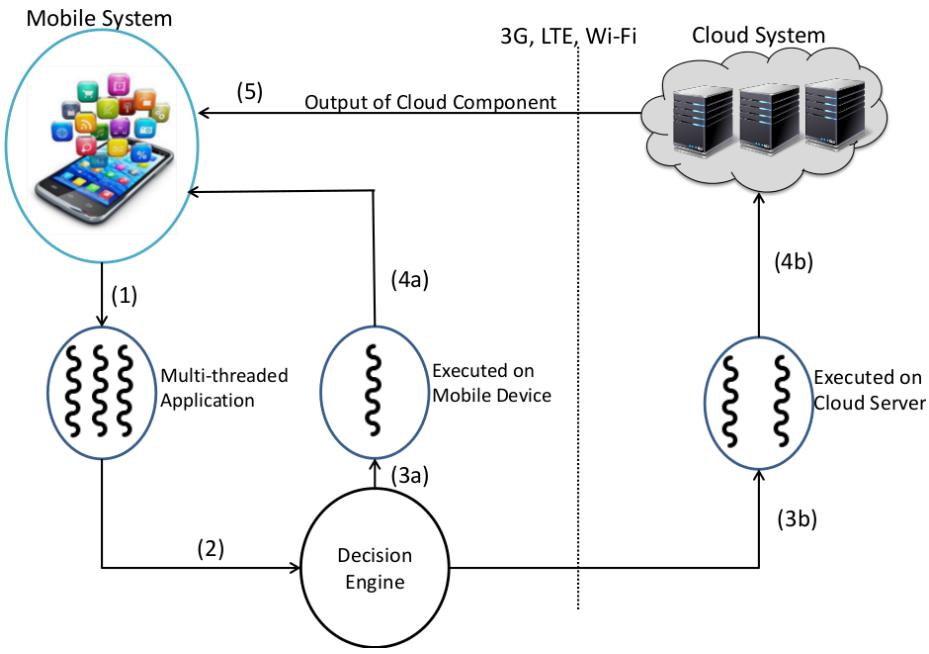

Fig. 1: Execution of Mobile Application using cloud server. One component of the application is executed on the mobile device, while the other component is executed on the cloud server. The offloading decision engine is typically executed on a separate server.

both the mobile device and the cloud server. Thus during execution only the program states need to be migrated to the cloud.

We assume that mobile applications have multiple threads. We model concurrent mobile applications using its call graph, which is a Directed Acyclic Graph (DAG) representing task invocations within the application. Each vertex in the DAG represents a task of the application, and each edge represents a dependency between two tasks. The set of tasks in the application is denoted by the vertex set $\mathbb{V}$, while the set of dependencies is represented by the edge set $\mathbb{E}$. Executing a task v_i locally on the mobile device incurs e_i^{loc} energy and t_i^{loc} time cost respectively. The application needs to be completed within a time deadline $\mathcal{D}$ and an energy budget $\mathcal{B}$. Some tasks, called native tasks, depend on mobile sensors and must always be executed on the mobile device.

Fig. 2 shows an application model where the application has three threads of execution. Two new threads are spawned at v_2. The threads join at v_7 and v_9 respectively. Moreover, three of the methods, v_1, v_4 and v_9 are native, i.e. they must be executed on the mobile device. This DAG model is general in nature, and can be used to model any mobile application.

The second component of an MCC offloading framework is the wireless network. Executing two tasks having dependency between them on different platforms (mobile or cloud) incurs a migration cost. Thus, if there exists an edge (v_i, v_j) to denote a dependency between tasks t_i and t_j, then this incurs a migration cost. This is represented

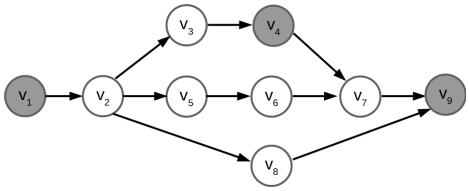

Fig. 2: A general Directed Acyclic Graph (DAG) representing a mobile application. Methods shaded gray are native, i.e. they must be executed on the mobile device.

$\mathbb{V}$	Vertex set of the call graph
$\mathbb{E}$	Edge set of the call graph
v_i	A method in the call graph
(v_i, v_j)	A call invocation of the task v_j by v_i
t_i^{loc}	Local time execution cost of each method v_i
e_i^{loc}	Local energy execution cost of each method v_i
e_{ij}^{mig}	Energy migration cost of the call invocation (v_i, v_j)
t_{ij}^{mig}	Time migration cost of the call invocation (v_i, v_j)
F	Speedup of the cloud compared to the mobile device
$\mathcal{D}$	Time deadline given to application
$\mathcal{B}$	Energy budget given to application

Table 1: Symbols introduced in Section 2

by e_{ij}^{mig} and t_{ij}^{mig} to represent migration energy and time respectively. We assume that these costs do not vary once execution of an application begins. This is a standard assumption used by all MCC offloading frameworks.

The third component of an MCC offloading framework is the cloud system. The cloud system has higher computing resources than the mobile device. We represent the ratio of the computing speed of the cloud to that of the mobile processor by F. Thus, the time cost of executing a task t_i on the cloud system is equal to t_i^{loc}/F. Moreover, execution on the cloud system incurs no computation energy cost on the mobile device.

3 Task Partitioning and Offloading: Formal Model

In this section, we formulate the offloading decision problem of a Mobile Cloud Computing (MCC) system for a mobile application. The task partitioning and offloading problem is NP-Complete for general concurrent applications [12]. Thus, we develop an integer-linear programming (ILP) problem to model this problem.

3.1 Problem Formulation

Let x_i be a binary decision variable such that:

$$x_i = \begin{cases} 1 & \text{if task } v_i \text{ is executed locally} \\ 0 & \text{if task } v_i \text{ is executed on the cloud} \end{cases}$$

x_i	Decision variable denoting execution location of v_i
st_i	Start time of executing the task v_i
l_i	Time taken to execute the task v_i
σ_{ij}	Decision variable denoting execution precedence
sm_{ij}	Start time of migration of the edge (v_i, v_j)
λ	Scaling factor used in optimization function

Table 2: Variables introduced in Section 3

Since there is a single decision variable to denote the location of execution of each method, every method in the call graph has to be executed (by choosing either $x_i = 0$ or $x_i = 1$).

Let the start time and execution duration of a task v_i be st_i and l_i respectively. Then, the completion time of a task is $st_i + l_i$. We know that all tasks must be completed by the given deadline $\mathcal{D}$. The time required for completing a task locally and on the cloud are t_i^{loc} and $t_i^{loc}/\mathcal{F}$ respectively.

Let σ_{ij} be a binary variable for all pair of tasks t_i and t_j such that:

$$\sigma_{ij} = \begin{cases} 1 & \text{if } v_i \text{ finishes execution before starting } v_j \\ 0 & \text{otherwise} \end{cases}$$

The variable σ_{ij} allows us to schedule the execution of tasks that have no dependencies between them in parallel.

Precedence constraint: We know that for all edges (v_i, v_j) in the graph, the task v_j has to be executed only after v_i has completed. This precedence constraint is represented using the variable σ_{ij}.

$$\forall (v_i, v_j) \in \mathbb{E},\ \sigma_{ij} = 1 \tag{1}$$

The nature of the above precedence constraint is such that if task v_i is executed after task v_j, then the opposite cannot be true. To enforce this property of precedence, we ensure that if for any such pair of tasks, if $\sigma_{ij} = 1$, then $\sigma_{ji} = 0$.

$$\forall v_i, v_j \in \mathbb{V},\ \sigma_{ij} + \sigma_{ji} \leq 1 \tag{2}$$

Concurrency constraint: First, we consider the case of a single processor on the mobile device. Thus, if the tasks v_i and v_j are scheduled by the offloading framework concurrently, i.e. $\sigma_{ij} = \sigma_{ji} = 0$, then at least one of the tasks must be executed on the cloud. In other words if $\sigma_{ij} = \sigma_{ji} = 0$, then at least one among x_i and x_j must be equal to 0. On the other hand, if both the tasks are executed locally, i.e. $x_i = x_j = 1$, then the two tasks must have some order between them.

$$\forall v_i, v_j \in \mathbb{V}, x_i + x_j \leq 1 + \sigma_{ij} + \sigma_{ji} \tag{3}$$

We have the following possible cases:

1. Tasks v_i and v_j have some order between them, i.e. $\sigma_{ij} + \sigma_{ji} = 1$. Then, both the tasks v_i and v_j may be executed either on the cloud or on the mobile device, and so x_i and x_j remain unconstrained.

2. Tasks v_i and v_j do not have any order between them, i.e. they may or may not be executed concurrently. If they are scheduled for concurrent execution, then at least one among v_i and v_j must be executed on the cloud. In this case it is possible to have $x_i = 0, x_j = 0$; $x_i = 0, x_j = 1$ or $x_i = 1, x_j = 0$. On the other hand, they may also be scheduled so that execution of one method commences only after the other finishes. In this case, the two methods may be executed at any point, i.e. both x_i and x_j can have any value, since $\sigma_{ij} + \sigma_{ji}$ is set to 1.

Extending this for n processors, we note that if $(n + 1)$ threads are scheduled for parallel execution, then at least one of them must be scheduled for execution on the cloud. To do so, we now pick all combinations of $(n + 1)$ methods from the DAG. The constraint can then be mathematically represented as:

$$\forall v_i, ..., v_{i_{n+1}} \in \mathbb{V}^{n+1}, \sum_{k=1}^{n+1} x_{i_k} \leq 1 + \sum_{(k,l) \in \mathbb{V}^2} \sigma_{i_k i_l} \tag{4}$$

For each combination of $(n + 1)$ methods, we ensure that if the number of tasks being executed concurrently are higher than the number of processors on the mobile device, then one or more of the tasks are scheduled for execution on the cloud. In that case, LHS of Equation 4 has a value equal to $n+1$. Thus, the amount of concurrency too has to reduce suitably so that the RHS increases in value. It is possible that executing the tasks sequentially gives a lower objective value. Then, the LHS of Equation 4 has a lower value.

We note that Equation 3 is a special case of Equation 4 corresponding to the case of a single mobile processor. This is because, by setting $n = 1$ in Equation 4, we get:

$$\forall v_{i_1}, v_{i_2} \in \mathbb{V} \times \mathbb{V}, x_{i_1} + x_{i_2} \leq 1 + \sigma_{i_1 i_2} + \sigma_{i_2 i_1} \tag{5}$$

Setting i_1 as i and i_2 as j in Equation 5, we get Equation 3.

Execution Time constraint: Executing a method v_i takes t_i^{loc} time if done locally on the mobile device, and t_i^{loc}/F on the cloud. Before commencing execution, output from all tasks v_j that immediately precede v_i, i.e. all possible v_j such that $(j, i) \in \mathbb{E}$, have to be migrated to the location where v_i is executed. The time required for this migration must be considered along with the actual execution time. Migrating a task requires the time needed to bring all the data from its preceding tasks.

$$\forall v_i \in \mathbb{V},\ l_i = x_i t_i^{loc} + (1 - x_i) t_i^{loc}/F + \sum_{(j,i) \in \mathbb{E}} |x_i - x_j| t_{ij}^{mig}, \tag{6}$$

where t_{ij}^{mig} refers to the migration time between the edges (v_i, v_j). The migration time depends only on the data transfer d_{ij}, which is fixed for a particular edge. Since this formulation assumes constant bandwidth, the time cost of migration t_{ij}^{mig} is a constant.

The first two terms of the above constraint refers to the computation time locally and on the cloud respectively, whereas the last term refers to the time required to migrate the data dependency. If v_2 is executed on the cloud, then $x_2 = 1$ and the constraint gives computation time as t_2^{loc}/F. Depending on where v_1 was executed, migration

cost might also have to be added to the computation cost of v_2 to get the total execution length of v_2.

Deadline constraint: The final task $v_{|\mathbb{V}|}$ has to complete execution before the given deadline.

$$st_{|\mathbb{V}|} + l_{|\mathbb{V}|} \leq \mathcal{D} \tag{7}$$

Energy budget constraint: The total energy consumption must not exceed the energy budget.

$$\sum_{i \in V} x_i e_i^{loc} + \sum_{(i,j) \in \mathbb{E}} |x_i - x_j| e_{ij}^{mig} \leq \mathcal{B} \tag{8}$$

Start time constraint: If method v_j is scheduled to execute after v_i (denoted by σ_{ij}), then the start time v_j is not less than the ending time of task v_i. Otherwise, we do not have any constraint on the start time of v_j, st_j. In that case, we reduce the right hand side of the constraint to a negative value to make st_j unconstrained. To do so, we use the largest time value in this formulation, which is the time deadline $\mathcal{D}$.

$$\forall v_i, v_j \in \mathbb{V}, st_j \geq st_i + l_i + (\sigma_{ij} - 1)\mathcal{D} \tag{9}$$

Finally, there can be two different objectives: minimizing energy consumption and minimizing execution time. The first objective, minimizing energy consumption, is:

$$\text{Min} \sum_{i \in V} x_i e_i^{loc} + \sum_{(i,j) \in \mathbb{E}} |x_i - x_j| e_{ij}^{mig} \tag{10}$$

This optimization function includes both migration energy and cost of local execution.

Similarly, the second objective, minimizing execution time can be written as:

$$\text{Min } st_{|\mathbb{V}|} + l_{|\mathbb{V}|} \tag{11}$$

Since the ending time already includes the time cost of migration, we do not need to explicitly add this to the optimization function of time.

Any one objective among energy or time can be chosen by an offloading framework for optimization. However, it is also possible to optimize both of them together by applying a suitable scaling factor (λ). The optimization function is then represented as:

$$\text{Min } \lambda(st_{|\mathbb{V}|} + l_{|\mathbb{V}|}) + (1 - \lambda)(\sum_{i \in V} x_i e_i^{loc} + \sum_{(i,j) \in \mathbb{E}} |x_i - x_j| e_{ij}^{mig}) \tag{12}$$

3.2 Limitations of the Formulation

Our formulation has a few limitations. Firstly, it assumes that network transmissions succeed eventually. Wireless networks are inherently lossy and have a probability of failure. We assume that retransmissions at lower layers of network stack hide much of the transmission failures. Hence, considering the probability of failure in this formulation is not expected to affect the results of our study.

Secondly, we assume that the energy and time costs of each task is fixed on the mobile device. Thus, we ignore the effect of user input on the energy and time costs. Since offloading is mostly used for computation-intensive tasks, user input does not significantly affect execution costs.

4 Simulation Results

In this section, we study the sensitivity of the offloading solutions to various parameters through separate simulation experiments. These parameters include both changes in the properties of the applications, and of the overall offloading system. These experiments demonstrate the impact of parameters on the performance of the offloading system.

4.1 Simulation Settings

The simulation parameters and their values are shown in Table 3. Unless explicitly mentioned, these are the parameter values used for the experiments. The execution time for each method was chosen randomly with uniform distribution between 100 ms and 500 ms. The limits were chosen based on the range of values obtained from the trace log files of real Android applications [13]. Each experiment was repeated 20 times to ensure that any bias in the values of a particular instance do not affect the overall result.

The energy consumption value for each method was chosen randomly between 1 J and 20 J following uniform distribution. Most offloading frameworks utilize an energy model to determine at run-time the energy gain. If the system is heterogeneous and utilizes frequency scaling, then there is no direct correlation between execution time and energy consumption [14, 15]. Thus, taking random values of both execution time and energy consumption for each method is a reasonable assumption.

Parameter	Range of Values
Local execution time of each method (t_i^{loc})	100-500 ms
Local energy consumption of each method (e_i^{loc})	1-20 J [14, 15]
Data transferred for migration (d_{ij})	50 - 500 KB [4]
Energy for migration (e_{ij}^{mig})	$0.007d_{ij} + 0.005t_{ij}^{mig} + 5.9$ J [16]
Bandwidth	1 Mbps [4]
Round-trip Time or propagation delay (RTT)	70 ms [4]
Speed of cloud compared to mobile device (F)	10 [1]
Number of threads spawned from a particular method	0-2
Number of methods in each graph	20
Proportion of native methods in application call graph	30%
Number of experiments performed on each graph	20
Number of processors in mobile device	1

Table 3: Parameters used in simulation. These parameter values are used for all experiments, unless otherwise stated.

The size of data to be migrated during offloading is also required. To obtain the size of heap objects that have to be migrated, we refer to the work by Yang et al. [4]. The data transfer size varies between 50 KB and 500 KB.

The energy consumption of the network interface is calculated based on the energy model described by Balasubramanian et al. for a Wi-Fi interface [17]. In this energy model, the energy cost of data transfer is obtained as $0.007 \times d_{ij} + 0.005t_{ij}^{mig} + 5.9J$, where d_{ij} is the number of kilobytes transferred and t_{ij}^{mig} is the total time required for migration. This cost includes the energy required to activate the wireless card, and connecting the device to the access point.

4.2 Performance Evaluation

We study the gains achieved by the use of MCC systems in terms of either energy consumption or execution time. We measure the gain in energy consumption by taking the ratio of energy consumption utilizing mobile cloud to that of energy consumption using local execution of the application:

$$\text{Gain in energy consumption} = \frac{\text{Energy consumption using cloud system}}{\text{Energy consumption without using cloud system}}$$

Similarly, the gain in execution time is given by:

$$\text{Gain in execution time} = \frac{\text{Execution time using cloud system}}{\text{Execution time without using cloud system}}$$

We solve the model derived for concurrent applications in Section 3 in these experiments. Based on the performance results, we can infer that the formulation used to model MCC systems works correctly.

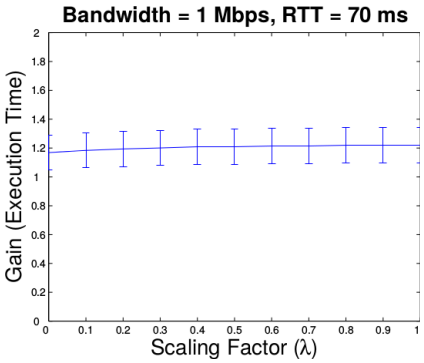

Fig. 3: Comparison of gain observed in execution time for ten different random graphs with increase in scaling factor (λ) along with the observed deviation from the mean.

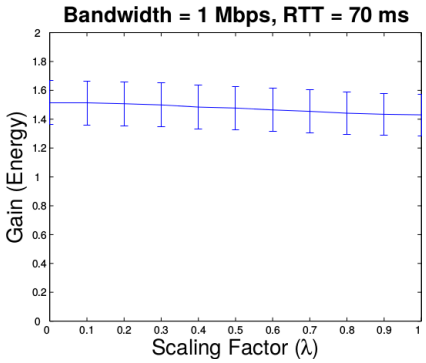

Fig. 4: Comparison of gain observed in energy consumption for ten different random graphs with increase in scaling factor (λ) along with the observed deviation from the mean.

The experiments are performed multiple times on different inputs to avoid any statistical error. We generate 10 graphs where the connectivity of the nodes, representing the function methods, is chosen randomly. Each graph has 20 nodes. The values of input parameters for each node, such energy consumed, and data transfer size for migration, are also varied randomly within the range described in Table 3. The average gains in execution time and energy consumption are then calculated based on the results derived from the 10 experiments on the 10 application call graphs generated.

Fig. 3 and 4 show the gains observed in execution time and energy consumption using our formulation. We note that the deviations observed from the mean are relatively

small (within a range of 0.2). This effectively confirms that the conclusions drawn from these results remain valid irrespective of the graph layout.

We observe that as the scaling factor (λ) used during optimization (Equation 12) increases, there is a small increase in gain (around 5%) observed in execution time. However, this comes at the cost of an increase in energy consumption. Thus, we conclude that execution time and energy consumption are conflicting objectives in some cases. An attempt to reduce the execution time might increase the energy consumption, and vice-versa.

The observation that execution time and energy consumption are conflicting objectives is explained by noting that a method having low execution time might consume a lot of energy. Thus, offloading such a method might end up increasing the execution time but reducing energy consumption. The opposite case, i.e. increasing the energy consumption but reducing the execution time due to offloading is also possible.

4.3 Impact of Application Variables

We investigate the effects of variabilities in different programs on the performance of MCC systems. Variabilities in a program could be due to difference in the number of native function calls, or the degree of parallelism in the code. Both the factors can be reflected in the graph representation of the program we have shown earlier. We study the effect of both of these factors on execution time and energy consumption.

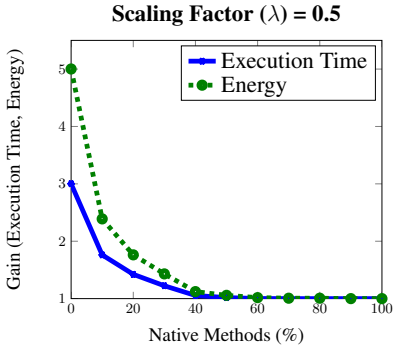

Fig. 5: Effect of increasing the number of native method in the application on execution time and energy consumption. Scaling factor in optimization is set to 0.5, i.e. equal priorities are given to time and energy optimization.

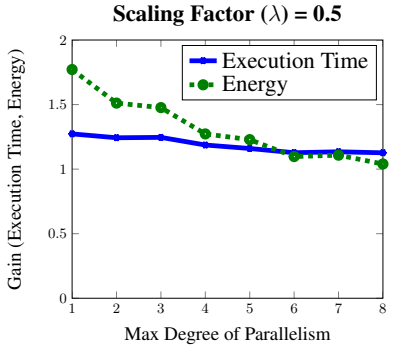

Fig. 6: Effect of increasing the maximum number of threads that can be spawned by a particular method. Scaling factor in optimization is set to 0.5, i.e. equal priority is assigned to time and energy optimization.

Effect of Native Methods: To study the impact of native methods, we gradually increase the probability of a method being native in the random graph. For each value of probability, we note the average gains in execution time and energy consumption.

Fig. 5 shows the effect of percentage of native methods on performance. We note that the increase in the percentage of native methods reduces the gain in both energy consumption and execution time. Moreover, when all the methods are native, the gains observed in both execution time and energy consumed are equal to 1.

These observations can be explained by noting that increasing the number of native methods forces more local execution of the application. This reduces the advantages of using the cloud. In the extreme case, when all the methods are native methods, then there is no gain in either energy consumption or execution time. This is expected, since the application executes locally and cannot take advantage of offloading.

We also observe that when the number of native methods is low, the reduction in performance with an increase in the number of native methods is non-linear. Thus, a small increase in the number of native methods leads to a very high drop in performance in terms of both execution time and energy consumption. This observation could be important for application developers trying to leverage the benefits of MCC.

This non-linear decrease in performance can be explained by observing that when the number of native methods is low, it is possible for the method that is spawning the threads itself to be migrated to the cloud. This avoids separate migration of multiple threads and therefore, reduces the cost of migration. Thus, very low number of native methods gives very high gains in both execution time and energy consumption.

Effect of Number of Threads spawned: To study the effect of number of threads spawned, we increase the maximum outgoing degree of each vertex. We have varied the maximum degree of each vertex from 1 to 8. We report the average gains for both execution time and energy consumption.

Fig. 6 shows the effect of increasing the number of threads spawned at each method of the application graph on performance. We observe that increasing the number of threads has almost no effect on execution time. However, the energy consumption involved increases with an increase in the number of threads.

To explain these observations, we note that increasing the number of threads increases both time and energy due to migration. However, the time spent on migration is mitigated by better utilization of parallelism. This effect does not apply to energy consumption, and so increases with an increase in the number of threads.

4.4 Detailed Study of Model Parameters

In order to understand the effect of individual environment parameters, we select a single representative graph using the layout shown in Fig. 2. This DAG is general in nature, and does not make any additional assumptions. It contains multiple threads with each of the threads spawned from the same method, but joins at different methods. Moreover, one of the threads also contains a native method. This ensures that all the different threads have conflicting requirements and thus, the decision problem becomes harder to solve.

Effect of Scaling Factor (λ): To study how the scaling factor affects the performance gains in this graph, we plot the energy and execution time for different values of the scaling factor. This result indicates how to balance the the two objectives, energy consumption and execution time, in the optimization objective function. Fig. 7 shows how varying the scaling factor affect both energy consumption and execution time.

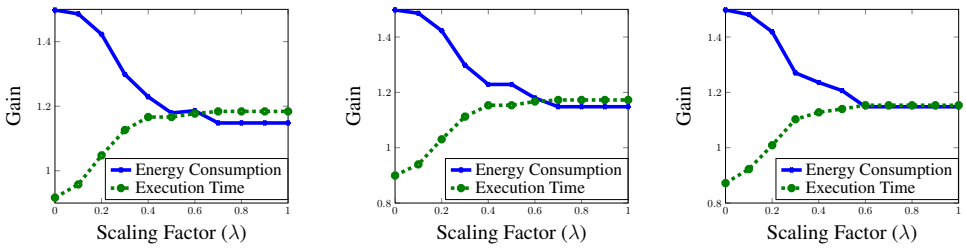

Fig. 7: Comparison of gain in energy consumption and execution time for different scaling factor values (λ). Round-trip Time (RTT) is used to measure cloud response time.

We observe that as the scaling factor increases, the total gain in execution time also increases. However, this comes at the cost of lower gains in energy consumption. When the scaling factor λ is set to 1, i.e. the optimization function considers only execution time, there is a speedup of 40% in execution time. However, there is no improvement in energy consumption. The opposite situation is observed when the scaling factor λ is set to 0. In this case the optimization function considers only energy, and so there is an improvement in energy consumption. This improvement in energy consumption comes at the cost of increased execution time as compared to local execution by around 10%.

This observation once again shows that in this graph too, execution time and energy consumption are conflicting objectives. Aggressively optimizing the execution time increases the energy consumption, and vice-versa. We have already explained the reasons behind this observation in Section 4.2.

We also observe that the gains in energy consumption and execution time are similar in all the three sub-figures. This means that the round-trip delay time does not affect the performance at this bandwidth. This observation can be explained by noting that at a bandwidth of 1 Mbps, most of the time is spent in transmission. Thus, the propagation delay is comparatively smaller, and hence does not affect performance.

Moreover, we also note that at a scaling factor of around 0.6, the gains in energy consumption and execution time are almost equal. This shows that irrespective of the cloud response time, a scaling factor equal to 0.6 balances both energy consumption and execution time. We explain this by noting that the conflicting requirements of time and energy are balanced when the scaling factor is equal to 0.6.

Effect of Speed of Cloud (F): We vary the speed of cloud ($\mathcal{F}$) as compared to the mobile device from 1 to 50. For each value of $\mathcal{F}$, we find the gain observed in execution time. We have studied the execution time for two cases – for very high and moderate bandwidths. Since speed of cloud does not have any effect on energy consumption, we have not included it in this study. Thus, the scaling factor (λ) has been set to 1 to ensure that only execution time is optimized by the system.

Fig. 8 shows the result of increasing the speed of cloud on execution time. We first note that due to utilization of parallelism, even a cloud system with very low speed gives an improvement of around 50% in execution time. At a low bandwidth, any improvement in the speed of cloud has very little effect on the total execution time. However, at

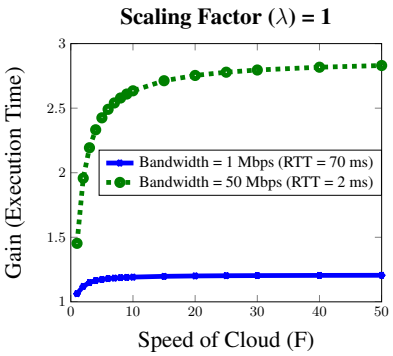

Fig. 8: Comparison of execution speed-up with increase in speed of the cloud system. Optimization function here considers only execution time i.e. scaling factor $\lambda = 1$

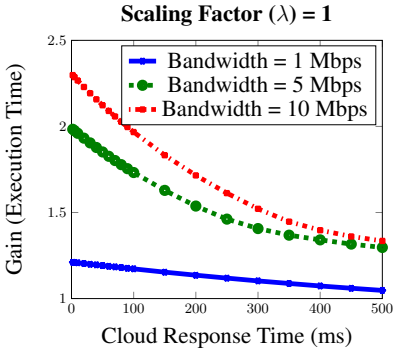

Fig. 9: Effect of round-trip delay (RTT) on execution time at different bandwidths. RTT is taken as an approximate measure of cloud response time.

high bandwidths, i.e. when migration time is low, the gain in execution time saturates at a much higher value of $F = 50$.

These observations can be explained by the fact that migration consumes more time than actual execution in case of moderate bandwidth. This explanation is further confirmed by the fact that at high bandwidth, much higher improvement in execution time is observed when the speed of cloud is increased. Further investigations on the effect of network bandwidth have been discussed later in this section.

Effect of Cloud Response Time: We now study the effect that propagation and transmission delays have on the total execution time (Fig. 9). We study how varying the cloud response time at three different bandwidths (1 Mbps, 5 Mbps and 10 Mbps) affect the execution time. Since the energy consumption remains same irrespective of the transmission and propagation time, we do not consider it here.

Fig. 9 shows the effect of increase in the propagation delay on execution time at the three different bandwidths. We observe that, at a low bandwidth of 1 Mbps, any increase in propagation delay has no effect on the execution time. However, this does not hold true at high bandwidths. At bandwidth of 10 Mbps, for instance, an increase in the RTT from 2 ms to 100 ms reduces the execution time by 20%.

This result can be explained by the fact that at high bandwidths, the propagation delay is higher than the actual transmission time during migration. However, at low bandwidths, the transmission time is much higher, and so most of the time is taken up by transmission. Thus, increasing the value of response time has no effect on execution time at low bandwidths, but has an adverse effect at high bandwidths.

Effect of Parallelism on Execution Time: We now investigate the gain on speedup with an increase in parallelism. Some offloading frameworks such as MAUI [1] do not exploit any parallelism in order to have a simpler mathematical formulation. Our objective is to determine if utilizing parallelism leads to any significant improvement in overall execution time.

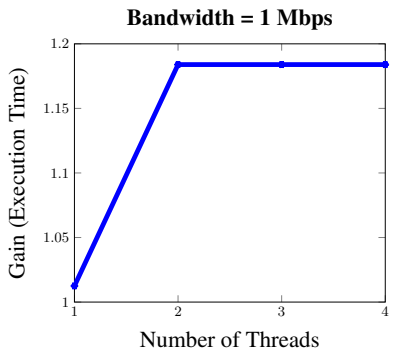

Fig. 10: Execution speedup measured with respect to increasing parallelism. We increased the number of threads that can run in parallel to measure the speedup in execution.

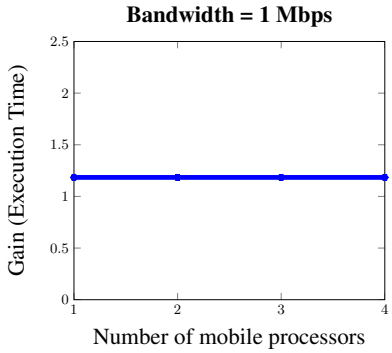

Fig. 11: Execution speedup measured with respect to increasing number of processors on the mobile device. λ denotes scaling factor used in optimization function.

In our first experiment, we increase the parallelism that can be utilized by the overall (mobile and cloud) system. For example, if the total number of threads is equal to 1, this implies that at a particular point of time, a total of 1 thread is executed (either locally at the mobile device or on the cloud). For more than 1 threads, since the mobile device has a single processor, the rest of the threads must execute on the cloud, if any parallelism is utilized. Once again, we do not consider the energy consumption. This is because according to our energy model, the energy consumption does not depend on the amount of parallelism used.

Fig. 10 shows how increasing the number of threads affects the execution time. Increasing the number of threads from 1 to 2 leads to an improvement of 45% in execution time. However, increasing the number of threads from 2 to 3 only leads to an improvement of 2% in execution time. Further increase in the number of threads leads to no improvement.

These observations can be explained by noting that our example graph has three parallel threads. Hence increasing the number of threads to greater than three has no effect on performance. Moreover, the third thread has a native method which must be executed locally on the mobile device. Thus, offloading this thread may or may not lead to any improvement in execution time. Hence the average gain observed when increasing the number of threads from 2 to 3 is small. However, since two of the threads do not contain any native methods, increasing the number of threads from 1 to 2 leads to a large improvement in execution time.

An alternative way of exploiting parallelism is to increase the number of processors in the mobile device itself. Once again, we study the increase in execution time when the number of mobile processors is increased. The result of our simulation is shown in Fig. 11. Our simulation result shows that this has no effect on the execution time. We explain this by noting that executing a thread on the cloud is usually faster as compared to local execution. Thus, even if a mobile processor is idle, the offloading framework

chooses to offload methods of a thread instead of scheduling it on the idle processor for local execution. Hence, increasing the number of processors on the mobile device shows no improvement in execution time.

5 Conclusion

Mobile cloud computing presents a solution to augment resource constrained mobile devices, where computationally intensive tasks can be partially offloaded to the cloud servers. Execution offloading to cloud helps in conserving computation energy on the mobile device, but consumes network energy to communicate with the cloud. Hence offloading decision must carefully select tasks to offload to eventually save energy on the mobile device. The task of offloading becomes more challenging due to the practical operating environment where there are multiple variable parameters. The effects of these parameters must be considered while making the offloading decisions.

In this work, we studied the impact of various parameters present in MCC systems on energy consumption and execution time of mobile applications. We presented a formal model of the offloading decision problem that incorporates various parameters that appear in real MCC execution environments. We utilize this model to study the impact of these parameters on the performance optimization objectives, like energy saved, and reduction in application execution time.

Acknowledgments

This research was supported by MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ICT Consilience Creative Program (IITP-2015-R0346-15-1007) supervised by IITP (Institute for Information & communications Technology Promotion).

Bibliography

[1] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir 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.

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[3] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen 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.

[4] Seungjun Yang, Yongin Kwon, Yeongpil Cho, Hayoon Yi, Donghyun Kwon, Jonghee Youn, and Yunheung Paek. Fast dynamic execution offloading for efficient mobile cloud computing. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 20–28. IEEE, 2013.

[5] Jiwei Li, Kai Bu, Xuan Liu, and Bin Xiao. Enda: Embracing network inconsistency for dynamic application offloading in mobile cloud computing. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, 2013.

[6] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. Cosmos: computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, pages 287–296. ACM, 2014.

[7] Weiwen Zhang, Yonggang Wen, Kyle Guan, Dan Kilper, Haiyun Luo, and Dapeng Oliver Wu. Energy-optimal mobile cloud computing under stochastic wireless channel. *Wireless Communications, IEEE Transactions on*, 12(9), 2013.

[8] Wei Gao, Yong Li, Haoyang Lu, Ting Wang, and Cong Liu. On exploiting dynamic execution patterns for workload offloading in mobile cloud applications. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 1–12, Oct 2014. doi: 10.1109/ICNP.2014.22.

[9] Marco V Barbera, Sokol Kosta, Alessandro Mei, Vasile C Perta, and Julinda Stefa. Mobile offloading in the wild: Findings and lessons learned through a real-life experiment with a new cloud-aware system. In *INFOCOM, 2014 Proceedings IEEE*, 2014.

[10] Y-D Lin, ET-H Chu, Y-C Lai, and T-J Huang. Time-and-energy-aware computation offloading in handheld devices to coprocessors and clouds. *IEEE Systems Journal*, 2013.

[11] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4), 1999.

[12] Tim Verbelen, Tim Stevens, Filip De Turck, and Bart Dhoedt. Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. *Future Gener. Comput. Syst.*, 29(2), 2013.

[13] Traceview. Profiling with traceview and dmtracedump. http://developer.android.com/tools/debugging/debugging-tracing.html, -.

[14] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 271–285, 2010.

[15] Kwang-Ting Cheng and Yi-Chu Wang. Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*. IEEE, 2011.

[16] Luis Corral, Anton B Georgiev, Alberto Sillitti, and Giancarlo Succi. Can execution time describe accurately the energy consumption of mobile apps? an experiment in android. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 31–37. ACM, 2014.

[17] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.