A Novel Offloading Partitioning Algorithm in Mobile Cloud Computing

Huaming Wu, Daniel Seidenstücker, Yi Sun, Carlos Martín Nieto, William Knottenbelt, and Katinka Wolter

Abstract—Mobile cloud offloading that migrates computation-intensive parts of applications from resource-constrained mobile devices onto remote resource-rich servers, is an effective way to shorten response time and extend battery life of mobile devices. Application partitioning plays a critical role in high-performance offloading systems, which involves splitting the execution of applications between the mobile side and cloud side so that the total execution cost is minimized. Through partitioning, the mobile device can have the most benefit from offloading the application to a remote cloud. In this paper, we study how to effectively and dynamically partition a given application into local and remote parts while keeping the total cost as small as possible. For general tasks (i.e., arbitrary topological consumption graphs), we propose a novel min-cost offloading partitioning (MCOP) algorithm that aims at finding the optimal partitioning plan (determining which portions of the application to run on mobile devices and which portions on cloud servers) under different cost models and mobile environments. The simulation results show that the proposed algorithm provides a stably low time complexity method and can significantly reduce execution time and energy consumption by optimally distributing tasks between mobile devices and cloud servers, and in the meantime, it can well adapt to environment changes.

Index Terms—Mobile device, mobile cloud computing, communication networks, offloading, cost graph, partitioning algorithm.

1 INTRODUCTION

Along with the maturity of mobile cloud computing, mobile cloud offloading is becoming a promising method to reduce execution time and prolong battery life of mobile devices. Its main idea is to augment execution through migrating heavy computation from mobile devices to resourceful cloud servers and then receive the results from them via wireless networks. Offloading is an effective way to overcome the computational and communication cost, and further derive a novel computational and communication graph (WCG) according to the estimated computational costs and edges reflect communication costs [1].

By partitioning the vertices of a graph, the calculation can be divided among processors of local mobile devices and remote cloud servers. Traditional graph partitioning algorithms (e.g., MAUI [2] and CloneCloud [3]) cannot be applied directly to the mobile offloading systems, because they only consider the weights on the edges of the graph, neglecting the weight of each node. Our research is situated in the context of resource-constrained mobile devices, in which there are often multi-objective partitioning cost functions, such as minimizing the total response time or energy consumption on mobile devices by offloading partial workloads to a cloud server.

In this paper, we explore the methods of how to deploy such an offloadable application in a more optimal way, by dynamically and automatically determining which parts of the application should be computed on the cloud server and which parts should be left on the mobile device to achieve a particular performance target (low latency, minimization of energy consumption, low response time, etc.) [3]. We study how to disintegrate and distribute modules of application between mobile devices and cloud server, and effectively utilize the cloud resources. The problem of whether or not to offload certain parts of an application to the cloud depends on the following factors: CPU speed of mobile device, network bandwidth, transmission data size, and the speed of the cloud server [4]. With considering these factors, we construct a weighted consumption graph (WCG) according to the estimated computational and communication cost, and further derive a novel...
min-cost offloading partitioning (MCOP) algorithm designed especially for the mobile offloading systems. This MCOP algorithm aims at finding the optimal cut that minimizes a given objective function (response time, energy consumption or the weighted sum of time and energy) and can be applied to WCGs of arbitrary topology.

The remainder of this paper is organized as follows. We review related work in Section 2. Section 3 explores the partitioning challenges and processes. Section 4 brings in the partitioning models such as topology, optimization and partitioning cost models. An optimal partitioning algorithm for arbitrary topology is proposed and investigated in Section 5. Section 6 describes three different profilers that are used for information collecting. Section 7 gives some evaluation and simulation results. Finally, the paper is summarized in Section 8.

2 RELATED WORK

Offloading becomes an attractive solution for meeting response time requirements and extending battery lifetime on mobile systems as applications become increasingly complex [10]. Karthik et al. [11] argued that offloading could potentially save energy and reduce execution time for mobile users, but not all applications are energy-efficient and time-saving when they are migrated to the cloud. It depends on whether the computational cost saved due to offloading outperforms the extra communication cost. A large amount of communication combined with a small amount of computation should preferably be performed locally on the mobile device, while a small amount of communication with a large amount of computation should preferably be executed remotely.

Many research efforts have been devoted to computation partitioning in mobile computing, in order to shorten response time or save energy consumption.

Compared with offloading a whole application into cloud, a partitioning scheme is able to achieve a fine granularity for computation offloading [12]. A partitioning algorithm introduced in [13] aims at reducing the response time of tasks on mobile devices. It finds the offloading and integrating points on a sequence of calls by depth-first search and a linear time searching scheme, and can achieve low user-perceived latency while largely reduce the partitioning computation on cloud. Some application partitioning solutions [14], [15], [16] heavily depend upon programmers and middleware to partition the applications, which limits their uses. Hence, automatic application partitioning like [17], [18] attracts more attention. The offloading inference engine proposed in [17] can adaptively make decisions at runtime, dynamically partition an application and offload part of the application execution to a powerful nearby surrogate.

Partitioning technologies were adopted to identify offloaded parts for energy saving [19], [20], [21]. The energy cost of each function of the application was profiled. According to the profiling result, they constructed a cost graph, in which each node represented a function to be performed, and each edge indicated the data to be transmitted. Finally, the server parts were executed on remote servers for reducing energy consumption. CloneCloud [3] used a combination of static analysis and dynamic profiling to partition applications automatically at a fine granularity while optimizing energy usage for a target computation and communication environment. However, this approach only considers limited input/environmental conditions in the offline pre-processing and needs to be bootstrapped for every new application built [22]. Due to frequent bandwidth fluctuations in the mobile environment, static application partitioning [18], [23], [24] cannot work well on mobile platforms. The availability of resources may change at the service nodes (available CPU power, memory, file cache, etc.) and at the wireless network (bandwidth, network latency, etc) [25]. Thus, optimal partitioning decisions should be made dynamically at runtime to adapt to different operating conditions.

This work was motivated by the above interesting works to investigate the partitioning problem in mobile cloud computing environment, aiming at the different objects, e.g., minimum of the response time, minimum of the energy consumption, and minimum of weighted sum of time and energy. We explicitly considered the mobile nature of both user and application behaviors, and addressed how dynamic partitioning can address these heterogeneity problems by taking the bandwidth as a variable. Thus, we greatly extending prior work by considering dynamic partitioning of applications between weak devices and clouds, in order to better support applications running on diverse devices in different environments.

3 PARTITIONING PROBLEMS

Application partitioning is very important for designing an adaptive, cost-effective, and efficient offloading system. Some critical issues concerning the partitioning problem include:

- **Weighting**: when choosing an application task to offload, we need to scale the weights of each application task regarding its resource utilization, such as memory, processing time, and bandwidth utilizations [26]. The weights can vary for different mobile devices and in different running environments. Communication overhead is introduced by the remote communication between a mobile device and a cloud server.

- **Real-Time Adaptability**: partitioning algorithms should be adaptive to network and device changes. For example, an optimal partition for a high-bandwidth low-latency network and low-capacity client might not be a good partition for a high-capacity client with a bad network connection. Since the network condition is only measurable at run time, the partitioning algorithm should be a real-time online process [13].

- **Partitioning Efficiency**: making partitioning decisions for simple applications (e.g., an alarm clock) at real-time is not difficult, but for some complex applications (e.g., speech/face recognition) that contain a large number of methods [13], a highly efficient algorithm is required to perform real-time partitioning.

3.1 Application Partitioning Processes

To solve the above challenges, the workflow of environment-adaptive application partitioning processes is proposed in Fig. 1. It starts with profiling an application that can be split into multiple tasks, through static analysis and dynamic profiling technology [27]. We then construct a WCG of the mobile application as shown in Fig. 3(b). Based on cost models, an elastic partitioning algorithm is proposed to make a proper application partitioning. By calling such an algorithm, we can get preliminary partitioning results for response time or energy...
optimization. During the execution process of the application, if the mobile environment changes, and these changes meet or exceed a certain threshold, the application graph will be re-partitioned according to the new parameters. Therefore, it can ultimately realize the condition-aware and environment-adaptive elastic partitioning. Here in the context of a mobile environment, it includes mobile computing resources inside the device, a battery level, CPU, memory, etc., but also includes an external mobile environment, such as the network connection and the cloud’s speed. After partitioning, it then automatically offloads the distributed applications that require remote execution to a cloud server and performs the rest locally on the mobile device according to the partitioning results.

Therefore, the problem of whether or not to offload certain parts of an application to the cloud depends on the following factors: CUP speed of the mobile device, network bandwidth, transmission data size, and the speed of the cloud server [9]. When considering such factors, we construct a WCG according to the estimated computational and communication cost, and further derive a novel partitioning algorithm designed especially for the mobile offloading systems.

### 3.2 Classification of Application Tasks

Different applications emerge in a mobile device according to some process and each consists of several tasks. Since not all the application tasks are suitable for remote execution, they need to be weighed and distinguished as:

- **Unoffloadable Tasks**: some should be unconditionally executed locally on the mobile device, either because transferring relevant information would take tremendous time and energy or because these tasks must access local components (camera, GPS, user interface, accelerometer or other sensors etc.) [2]. Tasks that might cause security issues when executed on a different place should also not be offloaded (such as e-commerce). Local processing consumes the battery power of the mobile device, but there are no communication costs or delays.

- **Offloadable Tasks**: some application components are flexible tasks that can be processed either locally on the processor of the mobile device, or remotely in a cloud infrastructure. Many tasks fall into this category, and the offloading decision depends on whether the communication costs outweigh the difference between local and remote costs or not [10].

We do not need to take offloading decisions for unoffloadable components. However, as for offloadable ones, since offloading all the application tasks to the remote cloud is not necessary or effective under all circumstances, it is worth considering what should be executed locally on the mobile device and what should be offloaded onto the remote cloud for execution based on available networks, response time or energy consumption. The mobile device has to take an offloading decision based on the result of a dynamic optimization problem.

### 4 Partitioning Models

In this section, we will illustrate which assumptions are made, how weighted consumption graphs (WCGs) for different types of applications are constructed and how the optimization problem is defined.

#### 4.1 Classification of Topologies

Flexible partitioning granularity-based applications are not limited to a specific form. Previous work considers application partitioning at different levels of granularity: classes [28], objects [27], methods [2], components [12], [29], and threads [3]. Without loss of generality, we refer to application tasks in this paper. Application developers can choose the appropriate partition granularity according to different applications.

Construction of WCGs is critical for the application partitioning. A mobile application can be represented as a list of fine-grained tasks, formulating different topologies as depicted in Fig. 2 where each node reflects an application task, executed either locally at the mobile side or offloaded onto the cloud side for remote execution.

![Flowchart of an application partitioning process](image)

![Task-flow graphs for different topologies](image)

(a) **Only one active node**: representing an entire application (without partitioning). Such a topology is often adopted by previous full offloading schemes such as [3], [30], [31], [32], which can also be viewed as an example of the software as a service. In this case, the whole application is migrated to a remote server involving complete
transfer of code and program state to the server \[33\]. The main drawback of this solution includes inflexibility and coarse granularity.

(b) *Linear topology:* representing a sequential list of fine-grained tasks \[13\]. Each task is sequentially executed, with output data generated by one task as the input of the next one \[34\].

(c) *Loop-based topology:* a loop-based application is one in which most of the functionality is given by iterating an execution loop, such as all the online social applications, in which we model their processing with a graph that consists of a cycle \[35\].

(d) *Tree-based topology:* representing a tree-based hierarchy of tasks \[33\]. The node at the top of the tree is the application entry node (i.e., the main module).

(e) *Mesh-based topology:* representing a lattice-based topology of tasks, e.g., a Java example of face recognition as depicted in \[27\].

When compared with the scheme that offloads the whole application (i.e., Fig. 2(a)) into the cloud, an application partitioning scheme is able to achieve a fine granularity for computation offloading when partitioning a topological consumption graph (CG) between local and remote execution. Different partitions can lead to different costs, and the total cost incurred due to offloading depends on multiple factors, such as device platforms, networks, clouds, and workloads. Therefore, the application may have different optimal partitions for different mobile environments and workloads.

### 4.2 Construction of Weighted Consumption Graphs

There are two types of costs in the offloading systems: one is computational cost of running the application tasks locally or remotely (including memory cost, processing time cost, and so on) and the other is communication cost for the application tasks’ interaction (associated with movement of data and requisite messages). Even the same task can have a different cost on the mobile device and the cloud in term of execution time and energy consumption. As cloud servers usually execute much faster than mobile devices having a powerful configuration, it can save energy and improve performance when offloading part of the computation to remote servers \[36\]. However, when vertices are assigned to different sides, the interaction between them leads to extra communication costs. Therefore, we try to find the optimal assignment of vertices for graph partitioning and computation offloading by trading off the computational costs with the communication costs.

Call graphs are widely used to describe data dependencies within a computation, where each vertex represents a task and each edge represents the calling relationship from the caller to the callee. Figure 3(a) shows a CG example consisting of six tasks \[14\]. The computational costs are represented by vertices, while the communication costs are expressed by edges. We denote the dependency of an application’s tasks and their corresponding costs as a directed acyclic graph \(G = (V, E)\), where the set of vertices \(V = \{v_1, v_2, \ldots, v_N\}\) denotes \(N\) application tasks and an edge \(e(v_i, v_j) \in E\) represents the frequency of invocation and data access between nodes \(v_i\) and \(v_j\), where vertices \(v_i\) and \(v_j\) are neighbors. Each task \(v_i\) is characterized by five parameters:

- *type:* offloadable or unoffloadable task.
- \(m_i:\) the memory consumption of \(v_i\) on a mobile device platform,
- \(c_i:\) the size of the compiled code of \(v_i\),
- \(\text{in}_{ij}:\) the data size of input from \(v_i\) to \(v_j\),
- \(\text{out}_{ij}:\) the data size of output from \(v_j\) to \(v_i\).

We further construct a WCG as depicted in Fig. 3(b). Each vertex \(v \in V\) is annotated with two-cost weights: \(w(v) = < w_{\text{local}}(v), w_{\text{cloud}}(v) >\), where \(w_{\text{local}}(v)\) and \(w_{\text{cloud}}(v)\) represent the computational cost of executing the task \(v\) locally on the mobile device and remotely on the cloud, respectively. Each vertex is assigned with one of the values in the tuple depending on the partitioning result of the application graph it finally ends up in or the label it is assigned \[37\]. The edge set \(E \subset V \times V\) represents the communication cost amongst tasks. The weight of an edge \(w(e(v_i, v_j))\) is denoted as:

\[
    w(e(v_i, v_j)) = \frac{\text{in}_{ij}}{B_{\text{upload}}} + \frac{\text{out}_{ij}}{B_{\text{download}}},
\]

which is the communication cost of transferring the input and return states when the tasks \(v_i\) and \(v_j\) are executed on different sides, and it closely depend on the network bandwidths (upload bandwidth \(B_{\text{upload}}\) and download bandwidth \(B_{\text{download}}\) and the transferred data.

A candidate offloading decision is described by one cut in the WCG, which separates the vertices into two disjoint sets, one representing tasks that are executed on the mobile device and the other one implying tasks that are offloaded to the remote server \[38\]. Hence, taking the optimal offloading decision is equivalent to partitioning the WCG such that an objective function is minimized \[39\].

The red dotted line in Fig. 3(b) is one possible partitioning cut, indicating the partitioning of computational workload in the application between the mobile side and the cloud side. \(V_l\) and \(V_c\) are sets of vertices, where \(V_l\) is the local set in which tasks are executed locally at the mobile side and \(V_c\) is the cloud set in which tasks are directly offloaded to the cloud. We have \(V_l \cap V_c = \emptyset\) and \(V_l \cup V_c = \tilde{V}\). Further, \(E_{\text{cut}}\) is the edge set in which the graph is cut into two parts.

### 4.3 Cost Models

Mobile application partitioning aims at finding the optimal partitioning solution that leads to the minimum execution cost, in order to make the best tradeoff between time/energy savings and transmission costs/delay.

The optimal partitioning decision depends on user requirements/expectations, device information, network bandwidth, and the application itself. Device information includes the execution speed of the device and the workloads on it when the application is launched. If the device computes very slowly and the aim is to reduce execution time, it is better to offload more computation to the cloud \[40\]. Network bandwidth affects data transmission for remote execution. If the bandwidth is very high, the cost in terms of data transmission will be low. In this case, it is better to offload more computation to the cloud.

The partitioning decision is made based on the cost estimation (comparative and communication costs) before the
program execution. On the basis of Fig. 3(b), we can formulate the partitioning problem as:

\[
C_{\text{total}} = \sum_{v \in V} I_v \cdot w_{\text{local}}(v) + \sum_{v \in V} (1 - I_v) \cdot w_{\text{cloud}}(v) + \sum_{e \in E} I_e \cdot w(e(v_i, v_j)),
\]

(2)

where the total cost is the sum of computational costs (local and remote) and communication costs of cut affected edges.

The cloud server node and the mobile device node must belong to different partitions. One possible solution for this partitioning problem will give us an arbitrary tuple of partitions from the vertices set \( < V_l, V_c > \) and the cut of edge set \( E_{\text{cut}} \) in the following way:

\[
I_v = \begin{cases} 
1, & \text{if } v \in V_l \\
0, & \text{if } v \in V_c
\end{cases} \quad \text{and} \quad I_e = \begin{cases} 
1, & \text{if } e \in E_{\text{cut}} \\
0, & \text{if } e \notin E_{\text{cut}} \end{cases}.
\]

(3)

We seek to find an optimal cut in the WCG such that some application tasks are executed on the mobile side and the remaining ones on the cloud side. The optimal cut maximizes or minimizes an objective function and meanwhile satisfies a mobile device’s resource constraints. The objective function expresses the general goal of a partition, this may be, for instance, minimize the energy consumption, minimize the amount of exchanged data, or complete the execution in less than a predefined time. We only actually perform the partitioning when it is beneficial. Not all applications can benefit from partitioning because of application-specific properties. The cost estimation of running each application task on the mobile device and cloud server is needed. Offloading makes sense only if the speedup of the cloud server overweigh the extra communication costs.

The communication time and energy costs for the mobile device will vary according to the amount of data to be transmitted and the wireless network conditions. According to (2), the dynamic execution configuration of an elastic application can be decided based on some different saving objectives with respect to response time and energy consumption. A task’s offloading goals may change due to a change in environmental conditions.

4.3.1 Minimum Response Time

The communication cost depends on the size of data transfer and the network bandwidth, while the computational cost is impacted by the computation time. If the minimum response time is selected as the objective function, we can calculate the total time spent due to offloading as:

\[
T_{\text{total}}(I) = \sum_{v \in V} I_v \cdot T_v^l + \sum_{v \in V} (1 - I_v) \cdot T_v^c + \sum_{e \in E} I_e \cdot T_e^r,
\]

(4)

where \( T_v^l = F \cdot T_c^l \) is the computation time of task \( v \) on the mobile device when it is executed locally; \( F \) is the speedup factor, the ratio of the cloud server’s execution speed compared to that of the mobile device, since the computation capacity of cloud infrastructure is stronger than that of the mobile device, we have \( F > 1 \); \( T_v^c \) is the computation time of task \( v \) on the cloud server when it is offloaded; \( T_e^r = D_e^r / B \) is the communication time between the mobile device and the cloud; \( D_e^r \) is the amount of data that is transmitted and received; \( B \) is the current wireless bandwidth.

In this scenario, the offloading decision engine then selects the best partitioning candidate that minimizes the total response time. The aim of this cost model is to find the optimal application partitioning \( I_{\min} = \{ I_v, I_e | I_v, I_e \in \{0, 1\} \} \) which satisfies \( I_{\min} = \arg \min_{I} T_{\text{total}}(I) \). For a given application and a mobile device, the optimal partitioning result also changes according to different wireless network bandwidth and speedup factor of the cloud server.

The saved response time in the partitioning scheme compared to the scheme without offloading is calculated as:

\[
T_{\text{save}}(I) = \frac{T_{\text{local}} - T_{\text{total}}(I)}{T_{\text{local}}} \cdot 100\%,
\]

(5)

where \( T_{\text{local}} = \sum_{v \in V} T_v^l \) is the local time cost when all the application tasks are executed locally on the mobile device.

4.3.2 Minimum Energy Consumption

If the minimum energy consumption is chosen as the objective function, we can calculate the total energy consumed due to offloading as:

\[
E_{\text{total}}(I) = \sum_{v \in V} I_v \cdot E_v^l + \sum_{v \in V} (1 - I_v) \cdot E_v^c + \sum_{e \in E} I_e \cdot E_e^r,
\]

(6)
where $E'_v = p_m \cdot T'_v$ is the energy consumed of task $v$ on the mobile device when it is executed locally; $E'_v = p_i \cdot T'_v$ is the energy consumed of task $v$ on the mobile device when it is offloaded to the cloud; $E_c = p_{tr} \cdot T'_e$ is the energy spent on the communication between the mobile device and the cloud; $p_m$, $p_i$, and $p_{tr}$ are the powers of the mobile device for computing, while being idle and for data transfer, respectively.

In this scenario, the offloading decision engine then selects the best partitioning plan that minimizes the partitioning cost of energy. The aim is to find the optimal application partitioning: $I_{\text{min}} = \{I_v, I_e | I_v, I_e \in \{0, 1\}\}$, which satisfies: $I_{\text{min}} = \arg \min_{I} E_{\text{total}}(I)$

The saved energy when compared to the scheme without offloading is:

$$E_{\text{save}}(I) = \frac{E_{\text{local}} - E_{\text{total}}(I)}{E_{\text{local}}} \cdot 100\%,$$

where $E_{\text{local}} = \sum_{v \in V} E'_v$ is the local energy cost when all tasks are executed locally.

### 4.3.3 Minimum of the Weighted Sum of Time and Energy

If we combine both the response time and energy consumption, we can design the cost model for partitioning as follows:

$$W_{\text{total}}(I) = \omega \cdot \frac{T_{\text{total}}(I)}{T_{\text{local}}(I)} + (1-\omega) \cdot \frac{E_{\text{total}}(I)}{E_{\text{local}}(I)},$$

where $0 \leq \omega \leq 1$ is a weighting parameter used to indicate relative importance between the response time and energy consumption. Large $\omega$ favors response time while small $\omega$ favors energy consumption. In some special cases performance can be traded for power consumption and vice versa. Therefore, the algorithm has $|V| - 1$ phases. In each phase $i$ (for $1 \leq i \leq |V| - 1$), the cut value, i.e., the partitioning cost in a graph $G_i = (V_i, E_i)$ is calculated. $G_{i+1}$ arises from $G_i$ by merging “suitable nodes”, where $G_1 = G$. The partitioning results are the minimum cut among all the cuts in an individual phase $i$ and the corresponding group lists for local and cloud execution. Furthermore, in each phase $i$ of the coarse partitioning, we still have five steps:

1. **Start with** $A = \{a\}$, where $a$ is usually an unoffloadable node in $G_i$.
2. **Iteratively add the vertex to $A$** that is the most tightly connected to $A$.
3. **Let $s, t$ be the last two vertices (in order) added to $A$**.
4. **The graph cut of the phase $i$ is** $(V_i \setminus \{t\}, \{t\})$.
5. **$G_{i+1}$ arises from $G_i$ by merging vertices** $s$ and $t$.

### 5 Partitioning Algorithm for Offloading

In this section, we introduce the min-cost offloading partitioning (MCOP) algorithm for WCGs of arbitrary topology. The MCOP algorithm takes a WCG as input which represents an application’s operations/calculations as the nodes and the communication between them as the edges. Each node has two costs: first is the cost of performing the operation locally (e.g., on the mobile phone) and second is the cost of performing it elsewhere (e.g., on the cloud). The weight of the edges is the communication cost to the offloaded computation. It is assumed that the communication cost between operations in the same location are negligible. The result contains information about the costs and reports which operations should be performed locally and which should be offloaded.

#### 5.1 Steps

The MCOP algorithm can be divided into two steps as follows:

1. **Unoffloadable Vertices Merging**: An unoffloadable vertex is the one that has special features making it unable to be migrated outside of the mobile device and thus is located only in the unoffloadable partition. Apart from this, we can choose any task to be executed locally according to our preferences or other reasons. Then all vertices that are not going to be migrated to the cloud are merged into one that is selected as the source vertex. By ‘merging’, we mean that these nodes are coalesced into one, whose weight is the sum of the weights of all merged nodes. Let $G$ represent the original graph after all the unoffloadable vertices are merged.

2. **Coarse Partitioning**: The target of this step is to coarsen $G$ to the coarsest graph $G_{[V]}$. To coarsen means to merge two nodes and reduce the node count by one. Therefore, the algorithm has $|V| - 1$ phases. In each phase $i$ (for $1 \leq i \leq |V| - 1$), the cut value, i.e., the partitioning cost in a graph $G_i = (V_i, E_i)$ is calculated. $G_{i+1}$ arises from $G_i$ by merging “suitable nodes”, where $G_1 = G$. The partitioning results are the minimum cut among all the cuts in an individual phase $i$ and the corresponding group lists for local and cloud execution. Furthermore, in each phase $i$ of the coarse partitioning, we still have five steps:

   a) **Start with** $A = \{a\}$, where $a$ is usually an unoffloadable node in $G_i$.
   b) **Iteratively add the vertex to $A$** that is the most tightly connected to $A$.
   c) **Let $s, t$ be the last two vertices (in order) added to $A$**.
   d) **The graph cut of the phase $i$ is** $(V_i \setminus \{t\}, \{t\})$.
   e) **$G_{i+1}$ arises from $G_i$ by merging vertices** $s$ and $t$.

#### 5.2 Algorithmic Process

The algorithmic process is illustrated as the MinCut function in Algorithm 2 and in each phase $i$, it calls the MinCutPhase function as described in Algorithm 3. Since some tasks have to be executed locally, we need to merge them into one node. The merging function is used to merge two vertices into one new vertex, which is implemented as in Algorithm 4. If nodes $s, t \in V$ ($s \neq t$), then node $s$ and node $t$ can be merged as follows:

1. **Nodes** $s$ and $t$ are chosen.
2. **Nodes** $s$ and $t$ are replaced with a new node $x_{s,t}$. All edges that were previously incident to $s$ or $t$ are now incident to $x_{s,t}$ (except the edge between nodes $s$ and $t$ when they are connected).
3. **Multiple edges** are resolved by adding edge weights. The weights of the node $x_{s,t}$ are resolved by adding the weights of $s$ and $t$. 
Algorithm 1 The 
Merging 
function

//This function takes s and t as vertices in the given graph and merges them into one
Function: G’ = Merge(G, w, s, t)

Input: G: the given graph, G = (V, E)
        w: the weights of edges and vertices
        s, t: two vertices in previous graph that are to be merged
Output: G’: the new graph after merging two vertices

1: x_{s,t} ← s ∪ t
2: for all nodes v ∈ V do
3:     if v /∈ {s, t} then
4:         w(e(x_{s,t}, v)) = w(e(s, v)) + w(e(t, v))
5:     //adding weights of edges
6:         \[ w_{\text{local}}(x_{s,t}), w_{\text{cloud}}(x_{s,t}) \] = \[ w_{\text{local}}(s) + w_{\text{local}}(t), w_{\text{cloud}}(s) + w_{\text{cloud}}(t) \]
7:     //adding weights of nodes
8:     E_c ≜ E ∪ e(x_{s,t}, v) //adding edges
9:     end if
10: \[ E_s', E_{t'} \] = \[ e(s, v), e(t, v) \] //deleting edges
11: end for
12: return G’ = (V’, E’)

Algorithm 2 The minCut function

//This function performs an optimal offloading partition algorithm
Function: \[ \text{minCut}(G, w, SourceVertices) \] = \[ \text{MinCut}(G, w, SourceVertices) \]

Input: G: the given graph, G = (V, E)
        w: the weights of edges and vertices
        SourceVertices: a list of vertices that are forced to be kept in one side of the cut
Output: minCut: the minimum sum of weights of edges and vertices among the cut
        MinCutGroupsList: two lists of vertices, one local list and one remote list

1: w(minCut) ← ∞
2: for i = 1 : length(SourceVertices) do
3:     //Merge all the source vertices (unoffloadable) into one
4:     (G, w) = Merge(G, w, SourceVertices(i), SourceVertices(i))
5:     end for
6: while |V| > 1 do
7:     \[ \text{cut}(A - t, t), s, t \] = MinCutPhase(G, w)
8:     if w(cut(A - t, t)) < w(minCut) then
9:         minCut ≜ cut(A - t, t)
10:     end if
11:     Merge(G, w, s, t)
12:     //Merge the last two vertices (in order) into one
13: end while
14: return minCut and MinCutGroupsList

Proof. As shown in Fig. [5], we use induction on the number of active vertices, k.

1) When k = 1, the claim is true,
2) Assume the inequality holds true up to u, that is C_{\text{cut}}(A_u, u) ≤ C_{\text{cut}}(H_u),
3) Suppose v is the first active vertex after u, according to the assumption C_{\text{cut}}(A_u, u) ≤ C_{\text{cut}}(H_u), then we have:

\[ \begin{align*}
C_{\text{cut}}(A_v, v) &= \text{C_{cut}(A_u, v)} + \text{C_{cut}(A_v - A_u, v)} \\
&\leq \text{C_{cut}(A_u, v)} + \text{C_{cut}(A_v - A_u, v)} (\text{u is MTCV}) \\
&\leq \text{C_{cut}(H_u)} + \text{C_{cut}(A_v - A_u, v)} \\
&\leq \text{C_{cut}(H_v)}. 
\end{align*} \]

Since t is always an active vertex with respect to H, by Lemma [6], we can conclude that C_{\text{cut}(A-t, t)} ≤ C_{\text{cut}(H)} which says exactly that the cost of cut(A - t, t) is at most as heavy as the cost of cut(H). This proves Theorem [7].
Algorithm 3 The MinCutPhase function

// This function perform one phase of the partitioning algorithm
Function: [cut(A−t,t), s, t]=MinCutPhase(G; w)

Input: G;: the graph in Phase i, i.e., Gi = (Vi, Ei)
w: the weights of edges and vertices
SourceVertices: a list of vertices that are forced to be kept in one side of the cut
Output: s, t: the lasted two vertices that are added to A
cut(A−t,t): the cut between {A−t} and {t} in phase i

1: a ∷ arbitrary vertex of G;  
2: A ∷ {a}  
3: while A ≠ Vi do  
4:     max = −∞  
5:     vmax = null  
6:     for v ∈ Vi do  
7:         if v ∉ A then  
8:             // Performance gain through offloading the task v to the cloud  
9:             ∆(v) = w(e(A, v)) − [w|local(v)| − w|cloud(v)|]  
10:            // Find the vertex that is the most tightly connected to A  
11:            if max < ∆(v) then  
12:                max = ∆(v)  
13:                vmax = v  
14:            end if  
15:        end if  
16:    end for  
17:    A = A ∪ {vmax}  
18:    a = Merge(G; w, a, vmax)  
19: end while  
20: t = the last vertex (in order) added to A  
21: s = the last second vertex (in order) added to A  
22: return cut(A−t,t)

Fig. 5. The illustration for the proof of Lemma 1

5.3 Computational Complexity

As the running time of the algorithm MinCut is essentially equal to the added running time of the |V| − 1 runs of MinCutPhase, which is called on graphs with decreasing number of vertices and edges, it suffices to show that a single MinCutPhase needs at most O(|V| log |V| + |E|) time yielding an overall running time. The computational complexity of the MCOP algorithm can be noted as O(|V|^2 log |V| + |V||E|).

As a comparison, linear programming (LP) solvers are widely used in schemes like [2] and [3]. The LP solver is based on branch and bound, which is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as general real valued problems [44]. The number of its optimal solutions grows exponentially with the number of tasks, which means higher time complexity O(2^|V|).

Therefore, the MCOP algorithm has much lower time complexity when compared to the existing algorithms, which is proportional to the square of the number of tasks and hence can achieve an optimal offloading strategy in minimal time.

5.4 Case Study

Figure 5 shows that node a is defined as the starting point in which the corresponding task will always be computed by the mobile device. We have s = d and t = f, and the induced ordering a, c, b, e, d, f of the vertices. Node f is cut off from the graph. The first cut-of-the-phase corresponds to the partitions {a, c, b, e, d} and {f}. Since the overall local cost is C|local| = ∑v∈V w|local(v)| = 45, we can calculate the cut cost by using Eq. (10) as: Ccut(A−f,f) = 45 − (15 − 5) + 5 = 40. At the end, we merge nodes s = d and t = f into one.

From Figs. 7-10 we repeat the same process of the MinCutPhase function as the first phase in Fig. 6. There are |V| − 1 = 5 phases, and at the end, all nodes are merged into one. Then, we compare all the cut values, the minimum value refers to the phase which has the optimal partitioning cut. In this scenario, the minimum cut of the graph G is the fourth cut-of-the-phase. The optimal cut is between {a, c} and {b, d, e, f} as depicted in Fig. 11 with the minimum cost of Ccut(a,c), (b, d, e, f) = 45 − (42 − 14) + (4 + 1) = 22. Here, tasks b, d, e, f are offloaded to the remote cloud server while tasks a and c are executed locally.

6 PROFILING

How to build the WCG is actually the bottleneck of whole technique, which closely depends on profiling, i.e., the process of gathering the information required to make offloading decisions. Such information may consist of the computation and communication costs of the execution units (program profiler), the network status (network profiler), and the mobile device specific characteristics such as energy consumption (energy profiler). Profilers are needed to collect information about the device and network characteristics, which is a critical part of the partitioning algorithm: the more accurate and lightweight they are, the more correct decisions can be made, and the lower overhead is introduced [22]. We will in the following introduce all types of profilers.

6.1 Program Profiler

A program profiler (static or dynamic) collects characteristics of applications, e.g., the execution time, the memory usage and the size of data. We can combine static analysis and dynamic profiling to construct the WCG of an application.
Fig. 6. The 1st phase of MinCutPhase function. The induced ordering of the vertices is a, c, b, e, s, t, where s = d and t = f. The 1st cut-of-the-phase corresponds to the partitions \{a, c, b, e, d\} and \{f\} with the cut value: \( C_{cut}(A,f) = 45 - (15 - 5) + 5 = 40. \)

Fig. 7. The 2nd phase of MinCutPhase function. The induced ordering of the vertices is a, c, b, e, s, t, where s = e and t = \{df\}. The 2nd cut-of-the-phase corresponds to the partitions \{a, c, b, e\} and \{d, f\} with the cut value: \( C_{cut}(A, (d,f)) = 45 - (27 - 9) + (1 + 3 + 4) = 35. \)

Fig. 8. The 3rd phase of MinCutPhase function. The induced ordering of the vertices is a, c, s, t, where s = b and t = \{def\}. The 3rd cut-of-the-phase corresponds to the partitions \{a, c, b\} and \{d, e, f\} with the cut value: \( C_{cut}(A, (d,e,f)) = 45 - (33 - 11) + (1 + 5) = 29. \)

Fig. 9. The 4th phase of MinCutPhase function. The induced ordering of the vertices is a, s, t, where s = c and t = \{bdef\}. The 4th cut-of-the-phase corresponds to the partitions \{a, c\} and \{b, d, e, f\} with the cut value: \( C_{cut}(A, (b,d,e,f)) = 45 - \{ (42 - 14) - (1 + 4) \} = 22. \)
Static analysis obtains the control flow graph of an application by analyzing the bytecode with nodes representing objects and edges representing relations between objects. We can get all the objects and the relations between them based on method invocations by traversing the graph. Constructing call graphs by hand and without the help of analysis tools would have cost far more time and resources. Many tools and frameworks have been developed to generate the call graph of a given application, e.g., Spark, Cgc, and Soot, and this automation is a huge advantage.

Dynamic profiling is adopted to obtain weights of the nodes and edges. Since there is a certain ratio of execution time to the total bytecode instruction count for Java programs, execution time of objects can be evaluated by the corresponding bytecode instruction count. Data transmission data between tasks include parameters and return values of method invocations. Combining Java bytecode rewriting with pretreatment information like speedup factor $F$ and wireless bandwidth $B$, we can obtain the execution time for each task (node weight) and the transmission time for each invocation (edge weight). These weights can be dynamically updated according to the varying processing capabilities of the cloud server and the wireless bandwidth.

We take a face recognition application as an example. By analyzing this application with Soot, the call graph could be constructed as a tree-based topology in Fig. 12. From the local estimated execution time, we can get the remote estimated execution time, dividing by the speedup factor $F$. When offloading a task to the cloud, the communication cost incurred between the mobile device and the cloud is the data transfer divided by the bandwidth. Then, we have the weighted consumption graph for this application. Finally, with remote execution and transmission costs, we now have all information to get the WCG.

### 6.2 Network Profiler

A network profiler collects information about wireless connection status and available bandwidth. It measures the network characteristics at initialization, and it continuously monitors environmental changes. Network throughput can be obtained by measuring the time duration when sending a certain amount of data as in [3]. Due to the mobile nature, the status of a wireless connection could change frequently (e.g., user moves to other location). Fresh information about a wireless connection is critical for the optimizer to make correct offloading decisions.

The profiler tracks several parameters for the WiFi and 3G interfaces, including the number of packets transmitted and received per second, and receiving and transmitting data rate. These measurements enable better estimation of the current network performance being achieved. We can use Speedtest to measure the mobile network bandwidth.

### 6.3 Energy Profiler

There are two ways to estimate the energy consumption, namely, software and hardware monitors. For example, MAUI used a power meter attached to the smartphone’s battery to build an energy profile. Power Monitor (e.g., Monsoon monitor) is a device that measures energy consumption when data is transmitted from the mobile device to the cloud server by supplying a certain level of power to the mobile device.

We can also use PowerTutor to measure the power consumption of the applications. Although PowerTutor does not give very accurate results as a hardware power monitor does, the result is still reasonable and does provide some values because it gives the detailed energy consumption information for each hardware component.

---

1. The face recognition application is built upon an open source code [http://darnok.org/programming/face-recognition/] which implements the Eigenface face recognition algorithm.

2. A free connection analysis tool, which shows real-time download and upload graphs, stores results both locally and on the Internet for sharing, [http://www.speedtest.net/]

3. PowerTutor is an application for Android phones that provides accurate, real-time power consumption estimates for power-intensive hardware components, [http://powertutor.org/]

---

Fig. 10. The 5th phase of MinCutPhase function. The induced ordering of the vertices is $s, t$, where $s = a$ and $t = \{b, c, d, e, f\}$. The 5th cut-of-the-phase corresponds to the partitions $\{a\}, \{b, c, d, e, f\}$ with cut value $C_{cut}(\{a\}, \{b, c, d, e, f\}) = 45 - (45 - 15) + 12 = 27$.

Fig. 12. Call graph of a face recognition application.
7 Evaluation of the Partitioning Algorithm

7.1 Setup
To evaluate the partitioning algorithm, we need to know three different kinds of values:

- **Fixed Values**: they are set by the mobile application developer, determined based on a large number of experiments. For example, the power consumption values of \( P_m, P_l, \) and \( P_t \) are parameters specific to the mobile system. We use an HP iPAQ PDA with a 400-MHz Intel XScale processor that has the following values: \( P_m \approx 0.9 \text{ W}, P_l \approx 0.3 \text{ W}, \) and \( P_t \approx 1.3 \text{ W} \) [11].
- **Specific Values**: such parameters represent some state of mobile devices, e.g., the size of transferred data, the value of current wireless bandwidth \( B \) (for convenient, we assume \( B_{\text{upload}} = B_{\text{download}} \)) and the speedup factor \( F \) that depends on the speed of current cloud server and the mobile device.
- **Calculated Values**: these values cannot be determined by application developers. For a given application, the computational cost is affected by input parameters and device characteristics, which can be measured using a program profiler. The communication cost is related to transmitting codes/data via wireless interfaces such as WiFi or 3G, which can be tracked by a network profiler.

Performance evaluation results encompass comparisons with other existing schemes, in contrast to the energy conservation efficiency and execution time. We compare the partitioning results with two other intuitive strategies without partitioning and, for ease of reference, we list all three kinds of offloading techniques:

- **No Offloading (Local Execution)**: all computation tasks of an application are running locally on the mobile device and there is no communication cost. This may be costly since as compared to the powerful computing capability at the cloud side, the mobile device is limited in processing speed and battery life.
- **Full Offloading**: all computation tasks of mobile applications (except the unoffloadable tasks) are moved from the local mobile device to the remote cloud for execution. This may significantly reduce the implementation complexity, which makes the mobile devices lighter and smaller. However, full offloading is not always the optimal choice since different application tasks may have different characteristics that make them more or less suitable for offloading [23].
- **Partial Offloading (With Partitioning)**: with the help of the MCOP algorithm, all tasks including unoffloadable and offloadable ones are partitioned into two sets, one for local execution on the mobile device and the other for remote execution on a cloud server node. Before a task is executed, it may require certain amount of data from other tasks. Thus, data migration via wireless networks is needed between tasks that are executed at different sides.

We define the saved cost in the partial offloading scheme compared to that in the no offloading scheme as **Offloading Gain**, which can be formulated as:

\[
\text{Offloading Gain} = 1 - \frac{\text{Partial Offloading Cost}}{\text{No Offloading Cost}} \cdot 100\%. \quad (11)
\]

The offloading gains in terms of time, energy and the weighted sum of time and energy are described in [5], [7] and [9], respectively.

7.2 Evaluation in Computational Complexity
We implement the MCOP algorithm in Java [4] that can serve as a comparison to the theoretic results. As an example, we partition the constructed weighted consumption graph in Fig. 12 under the condition of the speedup factor \( F = 2 \) and the bandwidth \( B = 1 \text{ MB/s} \) where the **main** and **checkAgainst** methods are assumed as unoffloadable nodes. The optimal partitioning result is depicted in Fig. [13]. The red nodes represent the application tasks that should be offloaded to the remote cloud and the blue nodes are the tasks that are supposed to be executed locally on the mobile device. The partitioning results will change as \( B \) or \( F \) varies.

![Fig. 13. Optimal partitioning result of the face recognition application when \( F = 2 \) and \( B = 1 \text{ MB/s} \).](https://github.com/carlosmn/work-offload)

The running time of the java implementation under different number of application tasks is depicted as Fig. [14]. We compare it with the theoretic computational complexity denoted as \( O(|V|^2 \log |V| + |V||E|) \) in Section 5.3. We find they have a good match with each other, which further proofs that our partitioning algorithm has much lower time complexity than the LP solver which has exponential time complexity.

![Fig. 14. Running time of the MCOP algorithm under different number of tasks](https://github.com/carlosmn/work-offload)
7.3 Evaluation in Dynamic Conditions

We build a graphical user interface (GUI) in MATLAB as shown in Fig. [15] The GUI is responsible for user interaction such as receiving input parameters and displaying the application partitioning results.

![Image of GUI](image)

Fig. 15. The user interface for demonstration

The user first inputs or selects the relative parameters, such as Application Graph, Unoffloadable Nodes and Optimization Model. We can either use the predefined application graphs of “linear”, “loop”, “tree” and “mesh” or just choose “user” to input any arbitrary consumption graph. Then, by clicking the “Graph” button, a weighted consumption graph will be constructed based on the above parameters. Further, by clicking the “Start Partitioning” button, the partitioning process will begin, by calling the partitioning algorithm of MCOP. We can get the partitioning results such as Partial Offloading Cost, No offloading Cost, Full Offloading Cost and Offloading Gain. In addition, the optimal partitioning graph will appear like Fig. [16] which further proves the correctness of the partitioning result in Fig. [11] with the minimum cost of 22. We can get different results under the different parameters of speedup factor \( F \) and wireless bandwidth \( B \).

![Image of optimal partitioning result](image)

Fig. 16. An optimal partitioning result of using the MCOP algorithm

In Fig. [17] the speedup factor is set to \( F = 3 \). Since the low bandwidth results in much higher cost for data transmission, the full offloading scheme can not benefit from offloading. Given a relatively large bandwidth, the response time or energy consumption obtained by the full offloading scheme slowly approaches to the partial offloading scheme because the optimal partition includes more and more tasks running on the cloud side until all offloadable tasks are offloaded to the cloud. With the higher bandwidth, they begin to coincide with each other and only decrease because all possible nodes are offloaded and the transmissions become faster. Both response time and energy consumption have the same trend as the wireless bandwidth increases. Therefore, bandwidth is a crucial element for offloading since the mobile system could benefit a lot from offloading in high bandwidth environments, while with low bandwidth, the no offloading scheme is preferred.

In Fig. [18] the bandwidth is fixed as \( B = 3 \) MB/s. It can be seen that offloading benefits from higher speedup factors. When \( F \) is very small, the full offloading scheme can reduce energy consumption of the mobile device, however it takes much more response time than the no offloading scheme. The partial offloading scheme that adopts the MCOP algorithm can effectively reduce execution time and energy consumption, while adapting to environmental changes.

From Figs. [17][18] we can tell that the full offloading scheme performs much better than the no offloading scheme under certain adequate wireless network conditions, because the execution cost of running methods on the cloud server is significantly lower than on the mobile devices when the speedup factor \( F \) is large. The partial offloading scheme outperforms the no offloading and full offloading schemes and significantly improves the application performance, since it effectively avoids offloading tasks in the case of large communication cost between consecutive tasks compared to the full offloading scheme, and offloads more appropriate tasks to the cloud server. In other words, neither running all tasks locally on the mobile terminal nor always offloading their execution to a remote server, can offer an efficient solution, but rather our partial offloading scheme can do.

In Fig. [19(a)] when the bandwidth is low, the offloading gain for all three cost models is very small and almost identical. That is because more time/energy will be spent in transferring the same data due to the low network bandwidth, resulting in increased execution cost. As the bandwidth increases, the offloading gain first rises drastically and then the increase becomes slower. It can be concluded that the optimal partitioning plan includes more and more tasks running on the cloud side until all the tasks are offloaded to the cloud when the bandwidth increases. In Fig. [19(b)] when \( F \) is small, the offloading gain for all three cost models is very low since a small value means very little computational cost reduction from remote execution. As \( F \) increases, the offloading gain first rises drastically and then approaches to the same value. That is because the benefits from offloading cannot neglect the extra communication cost. From Fig. [19] the proposed MCOP algorithm is able to effectively reduce the application’s energy consumption as well as its execution time. Further, it can adapt to environmental changes to some extent and avoids a sharp decline in application performance once the bandwidth decreases.

8 Conclusion and Future Work

In this paper, for applications under different scenarios, we construct them into different WCGs of arbitrary topology. To tackle the problem of dynamic partitioning in a mobile environment, we propose a novel offloading partitioning algorithm (MCOP
algorithm) that finds the optimal application partitioning under different cost models to make the best trade-off between time/energy savings and transmission costs/delay. Contrary to the traditional graph partitioning problem, our algorithm is not restricted to balanced partitions but takes the infrastructure heterogeneity into account.

The MCOP algorithm provides a stably quadratic runtime complexity for determining which parts of application tasks should be offloaded to the cloud server and which parts should be executed locally, in order to save mobile devices’ energy and to reduce application’s execution time. Experimental results show that according to environmental changes (e.g., network bandwidths and cloud server performance), the proposed algorithm can effectively achieve the optimal partitioning result.
in terms of time and energy saving. Offloading benefits a lot from high bandwidths and large speedup factors, while low bandwidths favor the no-offloading scheme.

Future work consists of integrating the MCOP with other algorithms (e.g., the one creates a graph out of program parts, the one partitions an application into parts and the one is able to prepare data of application parts that once offloaded, the cloud server is able to execute them) in an actual software deployment framework to automatically distribute software components on a cloud infrastructure.
[43] Y.-W. Kwon and E. Tilevich, “Energy-efficient and fault-tolerant distributed mobile execution,” in Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on, pp. 586–595, IEEE, 2012.

[44] Wikipedia, “Branch and bound.” http://en.wikipedia.org/wiki/Branch and bound.

[45] O. Lhoták and L. Hendren, “Scaling java points-to analysis using spark,” in Compiler Construction, pp. 153–169, Springer, 2003.

[46] “Soot: A framework for analyzing and transforming java and android applications,” in http://sable.github.io/soot/.

[47] W. Binder and J. Hulaas, “Using bytecode instruction counting as portable cpu consumption metric,” Electronic Notes in Theoretical Computer Science, vol. 153, no. 2, pp. 57–77, 2006.