Heuristic Offloading of Concurrent Tasks for Computation-Intensive Applications in Mobile Cloud Computing

Mike Jia, Jiannong Cao, Lei Yang
{csmjia, csjcao, csleiyang}@comp.polyu.edu.hk

Department of Computing - The Hong Kong Polytechnic University, Hong Kong

***Abstract*—Mobile applications are becoming increasingly computation-intensive, while the computing capacity of mobile devices is limited. A powerful way to reduce completion time of an application is to offload tasks to the cloud for execution. However, online offloading an application with general taskgraph is a difficult task. In this paper we present an online task offloading algorithm that minimizes the completion time of the application on the mobile device. We take cloud service time into account when making an offloading decision and we consider general taskgraphs for offloading. In our algorithm, for sequential tasks (i.e., line topology taskgraphs) we find the optimal offloading of tasks to the cloud. For concurrent tasks (i.e., general topology taskgraphs) we use a load-balancing heuristic to offload tasks to the cloud, such that the parallelism between the mobile and the cloud is maximized. Simulation results show that our algorithm has a performance of at least 85% of the optimal solution, and is significantly better than other existing algorithms.**

I. Introduction

Mobile devices nowadays are equipped with cameras, microphones and many other high quality sensors. Using these high-data rate sensory devices, it becomes possible for mobile devices to host perception related applications, such as face/gesture recognition, visual text translation, and video image processing. Many of these applications are computation-intensive, while the computing capacity on mobile devices is often limited. A powerful approach is to offload computation-intensive tasks to the cloud for execution [4], [6], [8], [11], [12]. However, it is a challenging issue to make an online decision about which tasks should be offloaded. Firstly, the combination of offloading decisions increases exponentially with the number of tasks which makes an exhaustive search unfeasible. Secondly, it is difficult to calculate the completion time of an application with a general taskgraph. This completion time depends on the parallelism of the mobile device and the cloud, the load of the cloud, and the network transmission between the mobile device and the cloud.

Recently, there has been some research studying the issue of task offloading. Odessa [9] is a typical example, they designed and implemented a run-time system that supports mobile applications offloading tasks to the cloud. They also presented a greedy offloading algorithm which incrementally makes a decision to offload each task in stages. While Odessa is fast, its offloading is far from optimal, and performs poorly when the network bandwidth is limited. A follow up work by L.Yang et al in [13] presents a similar framework for offloading general taskgraphs, and provides an offloading algorithm that maximizes throughput. However, the work in [13] failed to address cloud service time in their model which is not practical. In [15] Y. Zhang et al. presented an algorithm for partitioning computation intensive mobile applications. Although cloud service time was taken into account in their model, they only considered sequential invocations of function calls, which greatly simplified their problem, and limited the significance of their solution.

In this paper we present an online task offloading algorithm that minimizes the completion time of the application on the mobile device. We take cloud service time into account when making an offloading decision and we consider general taskgraphs for offloading. Our design is motivated by two observations: first, if a task is offloaded, it is likely that its neighboring tasks are also offloaded. Second, completion time of the application can be reduced by offloading tasks to the cloud in such a way that maximizes parallelism between the cloud and the mobile device. In our algorithm, for sequential tasks (i.e., line topology taskgraphs) we find the optimal offloading of tasks to the cloud. For concurrent tasks (i.e., general topology taskgraphs) we use a load-balancing heuristic to offload tasks to the cloud, such that the parallelism between the mobile and the cloud is maximized.

The rest of our paper is organized as follows. Section 2 contains reviews of related works. Section 3 explores the models we use in our problem formulation. Section 4 gives a detailed description of our task offloading algorithm and Section 5 presents our simulation results. A conclusion is drawn in Section 6.

II. Related Work

Mobile cloud computing is an emerging research topic and a great deal of research effort has been dedicated to it. Task offloading is an important technology that allows mobile users to take advantage of the powerful computing resources of the cloud. Many previous works have proposed task offloading frameworks and algorithms to optimize different performance metrics; for example, energy consumption on the mobile device [5], throughput of mobile data-streaming applications [13], and completion time [2], [3], [9], [7], [14]. Among them, CloneCloud [1], [2] and MAUI [3] are considered to be pioneering works in this area.

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

CloneCloud used an offline algorithm to precalculate the optimal offloading for an application given a set of parameters such as network data rate and the device computing capacity. Since it is an offline algorithm, the complexity is too high to be applied as an online algorithm. MAUI proposed an algorithm that made offloading decisions at runtime. It formulated its offloading problem as a 0-1 ILP, with each variable indicating whether the corresponding task is executed locally or on the cloud. The computation cost of solving the 0-1ILP makes it unsuitable as an online solution.

A follow-up work by Y. Zhang et al. in [15] presented an offloading algorithm that achieved near optimal offloading decisions in linear time. However their algorithm is only able to offload applications represented by a series of consecutive tasks. As they did not consider offloading concurrent tasks, the significance of their solution is limited.

The most similar work to ours is Odessa [9], which presented a greedy offloading algorithm for applications with general taskgraphs. It incrementally searches for the performance bottleneck in the system which can be a local task or a transmission link. Odessa then decides if the current performance can be improved by offloading the bottleneck task, or in the case of a bottleneck link, by offloading the local task of the bottleneck link. While Odessa is fast, its offloading is far from optimal, and performs poorly when the network bandwidth is limited. Furthermore, Odessa assumes that the cloud server is dedicated to the mobile user. In our work we propose a near-optimal online offloading algorithm, and consider the cloud server to be a shared resource among multiple application users.

III. System Model

The application is modeled by a directed taskgraph $G(V, E)$, where nodes in *V* represent tasks and edges in *E* represent the dependency of tasks. For task $i \in V$, there is a weight s_i, which is the average number of instructions of task *i*. A directed edge $(i, j) \in E$, represents a transition from task *i* to task *j*. We refer to task *i* as a parent task, and task *j* as a child task. A task with more than one child is called a fork task, and a task with more than one parent called a merging task. Each edge (i, j) is associated with a weight $d_{i,j}$ which represents the size of data transfer from task *i* to task *j*.

We assume that the mobile device has a single CPU with processing capacity μ_m. The radio set on the mobile device can work in parallel with of the CPU; that is, the data transmission between mobile device and the cloud can be done in parallel with CPU computation. The execution time m_i of task *i* is:

$$m_i = \frac{s_i}{\mu_m}. \tag{1}$$

We assume the cloud's processing capacity is μ_c. The cloud's computation power is typically an order of magnitude greater than that of the mobile device. However the cloud resource is shared among many application users, making the offloading decision dependent on the current load of the cloud. Let *Q* denote a set of tasks that are currently queued on the server of the cloud. The total queuing time of a newly arrived task is:

$$t_Q = \sum_{i \in Q} \frac{s_i}{\mu_c}. \tag{2}$$

The execution time of task *i* in the cloud (if task *i* is offloaded to the cloud) is:

$$c_i = \frac{s_i}{\mu_c}. \tag{3}$$

Note that c_i does not include queue time. When we offload a task to the cloud, the data transfer from its parent task on the mobile device becomes a network data transmission, and likewise for a task in the cloud transmitting data to its child task on the mobile device. Let *R* denote the data rate of the wireless communication between the mobile and the cloud. Given a link from task *i* to task *j*, if one of the tasks is on the mobile device and the other is on the cloud, the data transmission time for the link is:

$$t_{i,j} = \frac{d_{i,j}}{R}. \tag{4}$$

Our aim is to offload some tasks of an application to the cloud, to reduce the completion time of the application. We assume the application program is also hosted in the cloud. By offloading a task to the cloud, the only communication incurred between the mobile device and the cloud is the data transfer from one task to another. This is a common assumption made in previous works [9], [13], [15].

Our problem is, given the taskgraph $G(V, E)$ of an application, to find a set of tasks to be offloaded to the cloud such that the completion time of executing the application is minimized.

IV. Offloading tasks to minimize completion time

There are two types of task relationships: sequential tasks and concurrent tasks. For sequential tasks, we can reduce completion time by offloading tasks to the cloud as the cloud has a larger processing capacity than the mobile device. For concurrent tasks, in addition to taking advantage of the greater processing power of the cloud, we can also exploit the parallelism of the mobile device and the cloud. We will discuss the two cases separately in the following subsections.

A. Offloading Sequential Tasks

To reduce the completion time of a sequential taskgraph, we need to offload as many tasks to the cloud as possible to exploit the cloud's greater processing capacity. However, offloading a task incurs a network data transmission which needs to be less than the time saved by offloading the task, in order to reduce completion time. In some situations the network bandwidth can be so unfavorable that offloading tasks to the cloud will take even longer than running the task on the mobile device. It was proven in [15] that the optimal set of tasks to be offloaded in a sequential taskgraph would always be a sequence of consecutive tasks in the taskgraph. Taking taskgraph in Fig.1 as an example, the theorem in [15] states that if there are tasks that can be offloaded to the cloud to reduce the completion time, the tasks must be consecutive, that is, the tasks from *j* to *k* in Fig.1.

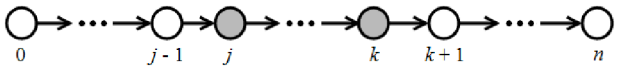

Figure 1. Offloading of Sequential Tasks

Algorithm 1 Offloading Sequential Task

in: $G(V, E)$
$EntryTask \leftarrow 1$;
$ExitTask \leftarrow 1$;
$T_{min} = T(1, 1)$;
  for $j \leftarrow 1$ *to* $n-1$ *//search for entryTask*
    for $k \leftarrow j$ *to* $n-1$ *//search for exitTask*
      if $T(j,k) < T_{min}$ then
        $EntryTask \leftarrow j$;
        $ExitTask \leftarrow k$;
        $T_{min} = T(j,k)$;
  end
out: $\{Task\, i \,|\, j \leq i \leq k\}$

For a sequence of *n* tasks in a line topology, assuming that the first and last tasks are performed locally, let the tasks to be offloaded to the cloud start from task *j* (the entry task) and end with task *k* (the exit task). Given the entry task *j* and the exit task *k,* the completion time is:

$$T(j,k) = \sum_{i=0}^{j-1} m_i + t_{j-1,j} + t_Q + \sum_{i=j}^{k} c_i + t_{k,k+1} + \sum_{i=k+1}^{n} m_i. \quad (5)$$

In the above formula, $\sum_{i=0}^{j-1} m_i$ is the total execution time for tasks on the mobile device before task *j*, $\sum_{i=j}^{k} c_i$ is the execution time of tasks offloaded to the cloud, $\sum_{i=k+1}^{n} m_i$ is the execution time of tasks after the offloaded tasks, and $t_{j-1,j}$ and $t_{k,k+1}$ represent the data transmission time to and from the cloud. Algorithm 1 shows the algorithm for finding the optimal entry and exit tasks, such that completion time is minimized.

B. Offloading Concurrent Tasks

Optimizing the completion time of concurrent tasks is much more challenging. For a taskgraph $G(V, E)$, we assume it begins with a fork task 0 and terminates with a merging task *n*. The first fork task and the last merging task are usually local tasks, because the first task (which we call the root task) needs to take inputs from the local device and the last task (which we call the termination task) needs to produce output back to the local device. We leave the discussion of general taskgraphs, where there are some sequential tasks before the root task, and some sequential tasks after the termination task, to the next subsection.

To offload concurrent tasks, our objective is to maximize parallelism between the mobile device and the cloud, which is equivalent to minimizing the completion time. Take Fig. 2 as an example, where there are *n* concurrent tasks between

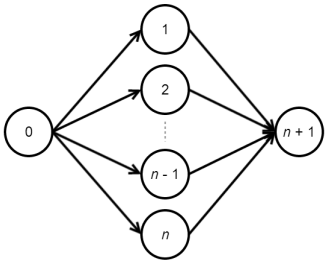

Figure 2. Concurrent Tasks

the root task 0, and the terminating task $n+1$. If we offload too many tasks to the cloud, then the mobile device must wait for the cloud to finish completing its share of the tasks. If we offload too few tasks to the cloud, then the application would wait for the mobile device to complete its tasks, which prolongs the completion time. The best case result of an offloading decision is where the waiting time at the termination task, between tasks executed on the cloud and tasks executed on the mobile device, is as small as possible. To achieve this goal, when computing task offloading, the completion time of the local tasks must be as close as possible to the completion time of the offloaded tasks plus the data transmission delays.

However, it is an NP-hard problem to determine which tasks should be offloaded for a general taskgraph. One important observation made by a previous work is that the offloaded tasks are always in clusters [15]. Once a task has been offloaded, the transmission cost of offloading additional neighboring tasks is greatly reduced, leading to clusters of tasks being offloaded. This is also true in a line topology, as the tasks offloaded are always in one contiguous segment. Finding clusters in a graph is very difficult. Our strategy is to reduce the graph into a tree. Then, we can identify clusters as subtrees and find the subtrees that should be offloaded.

1) Tree Generation: Converting the taskgraph into a tree is done by pruning the incoming links of each task, until each task only has one parent task. By reducing the taskgraph into a tree we can significantly reduce the complexity of the problem. However, there is a trade-off in information lost that may result in sub-optimal offloading decisions. To mitigate this, we prune the links with the lowest data loads to reduce the amount data being transmitted outside the scope of the tree. In Fig. 3(a) we have a concurrent taskgraph. Tasks 5, and 8 are merging tasks, and their incoming links have been labeled with their network transmission times. As $t_{3,5} > t_{2,5}$, and $t_{5,8} > t_{6,8} > t_{7,8}$, we remove links (2,5), (6,8), and (7,8) giving us the resulting task tree seen in Fig. 3(b).

2) Load Balancing Task Partitioning: Let V_C denote the set of tasks to be offloaded to the cloud, and V_M the set of tasks to be performed locally, so that $V_C + V_M = V - \{0, n\}$. We exclude the root task 0 and the termination task *n* from *V* when we considering tasks to offload.

Note that it was observed that tasks offloaded to the cloud are always in clusters. In a tree topology, this observation indicates that if a non-leaf task is offloaded, then all the

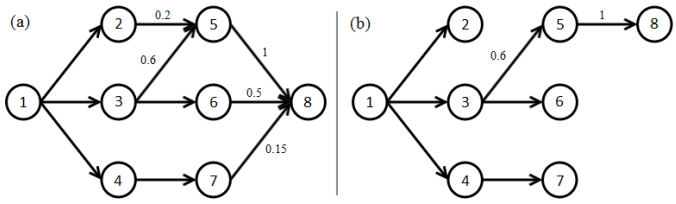

Figure 3. Tree Generation

tasks in the subtree rooted from this task should be offloaded together. This offloading decision is made at a fork task, where its children are the candidates for entry tasks. Let $tree(i)$ denote the set of tasks in the subtree rooted from task i (including i). We define M_i as the execution time of all tasks in $tree(i)$ on the mobile device:

$$M_i = \sum_{i \in tree(i)} m_i, \tag{6}$$

and C_i as the execution time of all tasks in $tree(i)$ on the cloud:

$$C_i = \sum_{i \in tree(i)} c_i. \tag{7}$$

We denote V_{entry} to be the set of entry tasks for offloading where each task in V_{entry} is the root of a subtree to be offloaded. As data transmissions only occur at the entry tasks in a tree topology, we define T_{net} to be the total transmission time between the cloud and mobile device:

$$T_{net} = \sum_{i \in V_{entry}} t_{parent(i),i}. \tag{8}$$

Let T_C be the execution time of tasks on the cloud:

$$T_C = \sum_{i \in V_M} c_i + \sum_{i \in V_{entry}} t_Q. \tag{9}$$

As tasks are offloaded to the cloud in subtrees, we assume that tasks in the same subtree are executed sequentially on the cloud without having to re-enter the cloud queue. This means we only need to count the queuing time of a cloud once per entry task. Finally we define T_M as the time taken to complete all tasks in on the mobile device:

$$T_M = \sum_{i \in V_M} m_i. \tag{10}$$

In order to minimize completion time of the application with concurrent tasks, we need to load balance tasks between the cloud and mobile device, so that the tasks on the mobile device finish in as close to the same time as the tasks on the cloud plus transmission time. Assuming the root task is on the mobile device, we have the following minimization objective:

$$\min |T_M - T_C - T_{net}|. \tag{11}$$

Initially V_C, is set to empty, which means all tasks are on the local device. We start from the root node, as shown in Fig.4. The children of the root are sorted in descending order of M_i defined in (6), where i is a child task of the root. We start to offload the subtrees rooted from these child nodes one by one, until $T_M < T_C + T_{net}$. We call the child node which makes this condition true for the first time, the break node. Note that all subtrees rooted from the child nodes before this break node should be offloaded to the cloud, and all the subtrees after this break node should be kept on the mobile device. This break node is a break even point. To achieve a more fine grain load balance, we need to further offload some of the sub-subtrees of this break node. We take this break node as the new root, and recursively offload the subtrees of the new root in the same way as described above. Each offloading decision at a sub-root will incrementally improve the load balance between the cloud and the mobile device. The algorithm terminates when the final break node is a leaf node. Algorithm 2 gives the details of our algorithm.

Algorithm 2 Offloading Concurrent Subgraph

in: $G(V, E)$

$r = \begin{cases} 0 & root\ task\ is\ on\ the\ cloud \\ 1 & root\ task\ is\ on\ the\ mobile \end{cases}$

$V_{net} = \emptyset$;

if r == 1 then $V_C = \emptyset$;

else $V_C = V - \{0, n\}$;

currentTask = root;

while root has children

$i \leftarrow 0$;

while $(2r-1)(T_M - T_C) > T_{net}$

$Task$ *child*$\leftarrow currentTask.getChild[i]$;

if r == 1 then $V_C \leftarrow V_C + tree(child)$;

else $V_C \leftarrow V_C - tree(child)$; end

$V_{entry} \leftarrow V_{entry} + child$;

$i \leftarrow i + 1$;

end

if r == 1 then $V_{entry} \leftarrow V_{entry} + child$;

else $V_C \leftarrow V_C - tree(child)$; end

$V_{entry} \leftarrow V_{entry} - child$;

// *roll back offloading for break task*

$currentTask = child$;

end

out: V_C

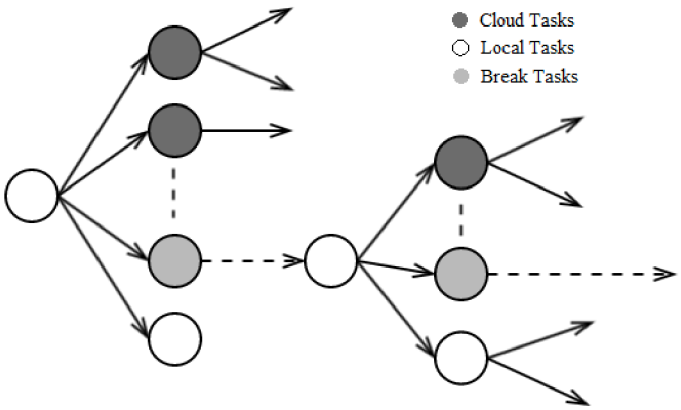

Figure 4. Tree Offloading

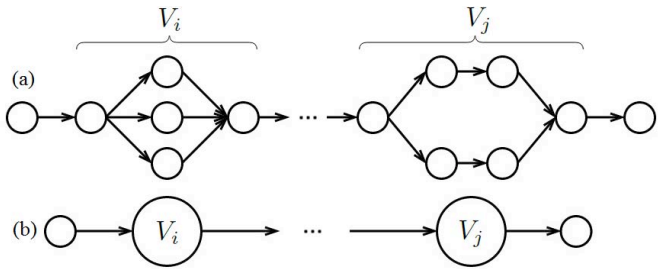

Figure 5. Example of a General Taskgraph

C. Offloading General Taskgraphs

In this subsection we describe how Algorithm 1 and 2 can be combined to provide a method for offloading general taskgraphs. A general taskgraph can be described as a sequence of linked sequential tasks and concurrent tasks. To offload a general taskgraph, we represent each set of concurrent tasks as a single virtual task, so that we have a line topology. Fig 5(a) is an example. Each concurrent subgraph in Fig. 5(a) is represented as a virtual node in Fig. 5(b). We define the computation weight of the virtual task as:

$$s_{V_i} = \sum_{j \in V_i} s_j. \tag{12}$$

After computing the offload for the line topology, some of the virtual nodes will be offloaded to the cloud and some may stay on the mobile device. In either case, we use algorithm 2 to compute the offloading of the concurrent tasks in the virtual nodes to take advantage of the parallelism between the mobile device and the cloud.

In the previous subsection, we used Formula (11) to describe the load balancing objective when the root task and termination task are executed locally. In the general case of offloading a concurrent subgraph, the root and the termination node of the subgraph may not be on the local device. We introduce a variable r, where $r = 0$ indicates the root is on the cloud, and $r = 1$ indicates the root is on the mobile device. We then revise our load-balancing objective in (11) to the general load balancing objective:

$$\min \left| (2r-1)\left(T_M - T_C\right) - T_{net} \right|. \tag{13}$$

When a concurrent subgraph has been offloaded, moving tasks back to the mobile device is the opposite of offloading tasks to the cloud. Algorithm 2 gives the details of our method.

V. Simulation

In this section we evaluate our proposed offloading algorithm. The performance metric we are concerned with is the completion time of the application taskgraph.

A. Methodology

First, we evaluate how the characteristics of a taskgraph can affect the completion time of the application. We then evaluate how the system environment parameters, such as bandwidth, cloud processing power, and cloud queuing time, affect performance.

	Taskgraph Parameters		Environment Parameters		
Set	*No. Tasks*	*Avg No. Out-links*	*Bandwidth*	μ_c	q
(a)	free	4	1	2.5	0.5
(b)	15	free	1	2.5	0.5
(c)	15	4	free	2.5	0.5
(d)	15	4	1	2.5	free

Table I
Simulation Configuration

To evaluate our algorithm we implemented a directed acyclic taskgraph generator based on the work in [10]. The taskgraphs are randomly generated using a level-by-level method to prevent any loops from occurring. A taskgraph has two characteristics which serves as in parameters in our taskgraph generator: the first is its size, which is the number of tasks, and the second is its link density, which is the average number of out-links at each task. The weights for each task and link are randomly sampled from a uniform distribution.

We have done a group simulation sets to evaluate the effect of both the taskgraph's characteristics and the system environment on the performance of our algorithm. In each simulation set we select a parameter as a variable, and keep all other parameters constant. We fix the local CPU $\mu_m = 1$ for all simulations. The completion time for each fixed value of a variable is the average completion time over 10 trial runs (each trial run has an independently generated taskgraphs with the same parameters). Table 1 shows the configuration of our simulation sets. 'Free' indicates that the parameter is a free variable in the simulation set.

In each simulation we compare the performance of our algorithm against two benchmarks, Odessa [9] and the optimal solution. The optimal results are found using the exhaustive search.

B. Results

Fig. 6(a) shows completion time against the number of tasks in the taskgraph. The performance of our algorithm is around 85-90% of the optimal. The performance of Odessa decreases steadily from 75% (5 tasks) to 45% (20 tasks). An explanation for why Odessa performs better on smaller taskgraphs is that the tasks offloaded by Odessa are more likely to be clustered together in a small taskgraph. As taskgraphs increase in size, the offloaded tasks are more scattered, which increases the amount of transmissions, and causes Odessa's performance to suffer.

In Fig. 6(b) we show completion time against average degree of out-links from each task. It appears that an increase in taskgraph link density has little effect on the completion time or the performance of our algorithm. However, Odessa's performance drops from 57% (average of 1 out-link) to 46% (average of 8 out-links). As the link density of the taskgraph increases, the transmission cost for offloading a task becomes greater, making it difficult for Odessa to find tasks to offload. This causes Odessa's performance decline as its completion time tends towards the completion time of an all-mobile offloading.

Fig. 6(c) shows completion time against bandwidth. All three plots show clear reciprocal curves as bandwidth in-

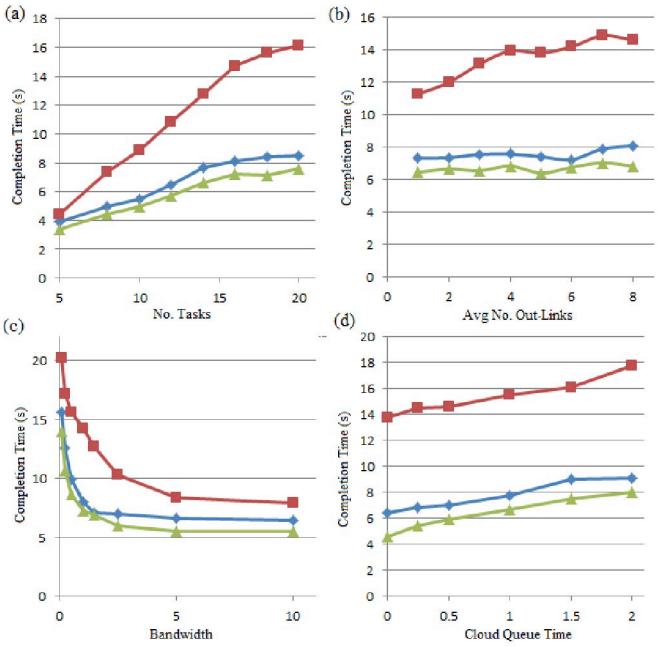

Figure 6. Simulation Results

creases. Our algorithm begins with a performance of around 90% of the optimal (bandwidth = 0.1) and increases to 95% (bandwidth = 1.5) before decreasing back to 85% (bandwidth = 10). The slight decline in performance between bandwidth 1.5 and 10, is possibly due to how our algorithm always leaves at least one sequential path of tasks on the mobile device, resulting in too few tasks being offloaded. Odessa begins with a performance of 70% of the optimal (bandwidth = 0.1), declining to 50% (bandwidth = 1) and increasing back up to 70% (bandwidth = 10). As bandwidth increases from 0.1 to 1, Odessa is able to offload more and more tasks and reduce completion time, however its performance compared to the optimal decreases in this period as Odessa does not offload enough tasks. As bandwidth continues to increases, Odessa eventually offloads all of its tasks, causing its performance catch up to 70% of the optimal.

Finally, Fig. 6(d) shows completion time against cloud queue time. Our algorithm has a performance between 85% to 90% compared to the optimal, and Odessa has a performance of 45% to 55%. As Odessa does not take cloud queuing time into consideration, it chooses to offload the same number of tasks on average, regardless of an increase in cloud queuing time, resulting in a near linear curve.

As has been observed, the completion time of our algorithm is consistently close to the optimal solution with a stable performance of at least 85%. The Odessa algorithm performs quite poorly in comparison, with a typical performance of 50%. Odessa has 2 main shortcomings that account for its performance. The first is that Odessa can only offload individual tasks one at a time, and is unable identify and offload clusters of tasks. As a result, Odessa frequently offloads too few tasks which stunts its performance, especially when the network bandwidth is limited. The second shortcoming is that Odessa fails to take advantage of parallelism between the cloud and the mobile device, and any parallelism that does occur is entirely incidental. Given a sufficiently large bandwidth, the Odessa algorithm will offload all the tasks to the cloud, as the transmission cost of offloading each task will be lower than the time difference between local and cloud execution. This leaves no room for parallel execution with the mobile device and prevents it from reaching the optimal solution, where parallelism between the cloud and mobile device is maximized. Our algorithm overcomes these shortcomings by offloading tasks in subtrees according to a load balancing heuristic, and as a result, achieves a near optimal performance.

VI. Conclusion

In this paper we presented an online offloading algorithm for general taskgraphs which minimizes completion time. Our algorithm used a load-balancing approach to offload tasks in clusters such that parallelism between the cloud and the mobile device is maximized. Simulation results have shown that the completion time of taskgraphs offloaded by our algorithm are twice as fast as those offloaded by the existing algorithm in Odessa [9], and has a performance of at least 85% when compared to the optimal solution.

References

[1] B. G. Chun, P. Maniatis. “Augmented Smartphone Applications Through Clone Cloud Execution,” - *HotOS 2009*

[2] B. G. Chun, P. Maniatis. “CloneCloud: Elastic Execution between Mobile Device and Cloud,” - *Proc. EuroSys’ 2010*

[3] E. Cuervo, A. Balasubramanian, D Cho. “MAUI: Making Smartphones Last Longer with Code Offload,” - *MobiSys’ 2010*

[4] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. “Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications,” - *Proceedings of Middleware 2009, pages 1-20. Springer, 2009.*

[5] K. Kumar, Y. Lu. “Cloud computing for mobile users: Can offloading computation save energy,” - *IEEE Computer Society 2010*

[6] Z. Li, C. Wang, R. Xu. “Task Allocation for Distributed Multimedia Processing on Wirelessly Networked Handheld Devices,” - *IPDPS 2002*

[7] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, S. Madden. “Wishbone: Profile-based Partitioning for Sensornet Applications,” - *NSDI 2009*

[8] S. Ou, K. Yang, and J. Zhang. “An effective offloading middleware for pervasive services on mobile devices,” - *Pervasive and Mobile Computing 2007*

[9] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, R. Govindan. “Odessa: Enabling Interactive Perception Applications on Mobile Devices,” - *ACM MobiSys’ 2011*

[10] T. Tobita, H. Kasahara. “A standard task graph set for fair evaluation of multiprocessor scheduling algorithms,” - *Journal of Scheduling 2002*

[11] R. Wolski et al. “Using Bandwidth data to Make Computation Offloading Decisions,” - *IPDPS 2008*

[12] K. Yang, S. Ou, H. H Chen. “On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications,” - *IEEE Communication Magazine 2008*

[13] L. Yang, J. Cao, Y.Yuan, T. Li, A. Han, A. Chan. “A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing,” - *ACM SigMetrics Performance Evaluation Review (PER) 2013*

[14] X. Zhang, A. Kunjithapatham, S. Jeong, S. Gibbs. “Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing,” - *Mobile Networks and Applications 2009*

[15] Y. Zhang, H. Liu, L. Jiao, X. Fu. "To offload or not to offload: an efficient code partition algorithm for mobile cloud computing," - *IEEE 1st International Conference on Cloud Networking 2012*