To offload or not to offload: an efficient code partition algorithm for mobile cloud computing

Yuan Zhang§, Hao Liu*, Lei Jiao§, Xiaoming Fu§
§Institute of Computer Science, University of Göttingen, Germany, {yuan.zhang, lei.jiao, fu}@cs.uni-goettingen.de
*Department of Computer Science, Tsinghua University, China, liuhao09@mails.tsinghua.edu.cn

Abstract— **A new class of cognition augmenting applications such as face recognition or natural language processing is emerging for mobile devices. This kind of applications is computation and power intensive and a cloud infrastructure would provide a great potential to facilitate the code execution. Since these applications usually consist of many composable components, finding the optimal execution layout is difficult in real time. In this paper, we propose an efficient code partition algorithm for mobile code offloading. Our algorithm is based on the observation that when a method is offloaded, the subsequent invocations will be offloaded with a high chance. Unlike the current approach which makes an individual decision for each component, our algorithm finds the offloading and integrating points on a sequence of calls by depth-first search and a linear time searching scheme. Experimental results show that, compared with the 0-1 Integer Linear Programming solver, our algorithm runs 2 orders of magnitude faster with more than 90% partition accuracy.**

Keywords-mobile cloud computing; code partition; algorithm; call graph; depth-first search

I. INTRODUCTION

A new class of cognition augmenting applications for smartphones such as face or object recognition, image or video editing, natural language processing and augmented reality is emerging and attracts increasing attention [1-3]. This kind of applications requires a large amount of computation, storage and energy which cannot be satisfied by off-the-shelf mobile devices [1-4]. While more sophisticated hardware can be adopted to carry out these application tasks, its usage will be limited by its size, weight and monetary cost. An alternative way of facilitating augmented cognition applications on mobile devices is to leverage cloud service or nearby infrastructure to support part of the execution. When the connection quality to the server is good, offloading some execution to the cloud could be beneficial. Given the ubiquitous deployment of WiFi/3G networks [14-15], the ongoing development of 4G systems [15], and the availability of commercial cloud infrastructure, the future of mobile cloud computing is promising.

When using the cloud infrastructure to execute mobile applications, a key issue is to determine what's the most energy/time- efficient way of allocating the components of the target application given the current network condition, which is also known as the code partition problem [6-9]. While cloud provides high computation capacity and energy savings, transferring the input and return state is time and energy consuming. In extreme cases, when the wireless connection is too weak or the amount of state needed to be transferred is too big, executing code on cloud can impair the performance and waste energy. Hence, before actually running its code on the cloud environment, an offloading decision should be made for the components of the target application with regard to the current network condition and device capacities. We follow the mobile cloud system architecture [6] shown in Fig. 1. As can be seen from the figure, the mobile cloud computing system usually contains two agents, one on the smartphone and one on the cloud. Code partition process is conducted before the real execution on the server based on the application identifier, device identifier and the RTT between the smartphone to the cloud. This kind of information provides the server with the knowledge of program complexity, hardware capacity and network conditions. After getting the partition layout, the smartphone would transfer the necessary state to enable remote execution. Even though a good partition layout can improve the performance, the decision making process itself is an overhead of the real computing. Thus, the code partition algorithm should be both accurate and fast. Furthermore, a good code partition algorithm should also have the following characteristics:

- Real time adaptability. The partition algorithm 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 bad network connection. Since the network condition is only measurable at run time, the code partition algorithm should be a real time online process.
- Partition efficiency. At the early stage of mobile cloud computing, researchers used to adopt the full Virtual Machine (VM) migration approach, in which the mobile device serves as a dumb client that receives and renders data [10-12]. This approach is gradually discarded by current designs [6-9], which generally take a more fine-grained perspective (typically at a method-level granularity) and only migrate computation intensive parts of the application. For simple applications (e.g., an alarm clock), making code partition decisions for these methods at real time is not difficult. However, some popular applications (e.g., speech/face recognition) are quite complex and contain many methods. Therefore, a highly efficient algorithm that can perform real-time code partition for

applications with a large number of methods is demanded.

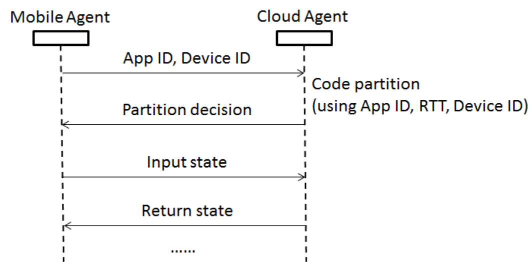

Figure 1. Process of mobile cloud compution and the role of code partition.

However, designing an accurate and efficient code partition algorithm for cognition augmenting applications is challenging. This kind of applications is usually sophisticated and contains a number of remoteable components [6] (i.e., methods that can be offloaded to the cloud). To better understand the characteristics of such applications, we downloaded 3 popular apps (Adobe Photoshop Express, Gesture Search, and Visidon AppLock) from Google Play [17-19]. We use reverse engineering tools [20-21] to disassemble the dex files and find that each of them contains more than 20 remoteable components. We also built two applications (one face recognition application[1], and one natural language processing application[2]) from open source code, and find that each of them contains also more than 15 remoteable components. The time required for solving the code partition problem by existing works with a 0-1 Integer Linear Programming (0-1 ILP) solver [6-7] is not negligible. For example, in the natural language processing application with 73 components the time needed for partition with 0-1 ILP solver is 7.3s compared to 21s time saving.

In this paper, we present an efficient code partition algorithm with high partition accuracy. Our design is motivated by the clustered offloading property of the call graph, which is based on the observation that when a method is offloaded, the subsequent invocations will be offloaded with a high chance. Hence, instead of making an offloading decision for each method as in the 0-1 ILP approach, our solution determines the best offloading and integrating points, which are several starting and ending methods pairs that would have a minimal cost if all the execution sequence in between are offloaded. We propose a $\Theta(V+E)$ time complexity code partition algorithm using depth-first search (DFS) and a linear time searching scheme, where V is the number of remoteable methods and E is the number invoking relations in the application. Experiment results show that our partition algorithm is 2 orders of magnitude speed up for the face recognition application and the natural language processing application than the 0-1 ILP approach with more than 90% accuracy.

[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] The natural language processing application is built upon an open source code http://nlp.stanford.edu/software/index.shtml

The rest of the paper is organized as follows. Section II reviews related works. Section III discusses the design rationale and the call graph model. Section IV describes the detailed design of our code partition algorithm. Section V presents the evaluation results compared with existing works. Conclusion and discussion are in section VI.

II. RELATED WORK

Several works have been proposed to tackle this code partition problem for mobile cloud offloading. Categorized by the granularity and partition algorithm been used, current work follows three approaches.

1) Full migration

Full migration [10-12] can also be viewed as an example of the software as a service, which is often adopted by early stage research. The inflexibility and coarse granularity of full migration are the main drawbacks of this approach.

2) Pre-calculated offline partitions

CloneCloud [8] follows this approach. It aims to profile as many execution paths as possible by alternating the inputs and then partitions each possible path under a few number of pre-defined conditions (two different network conditions (WiFi and 3G) for time metric, and eight states (<CPU, Scr, Net> triples) for energy metric). The smartphone searches for a matching partition in the database at run time. However, real network and device conditions cannot be generalized into fixed amount of states, and using the pre-calculated partitions cannot cover all the offloading scenarios.

3) Making decision at runtime

Examples of this approach are Odessa [9], MAUI [6] and Wishbone [7]. Odessa uses a greedy strategy by estimating whether to offload the current stage would be beneficial. Although the greedy algorithm is fast and can adapt to input and network changes, the decision is quite unreliable. MAUI and Wishbone formulate the partition problem into a 0-1 ILP where each variable is an indicator of whether the corresponding method should be offloaded. By using the previous profile of last run as an estimator of current run, the whole call graph can be partitioned with regard to the current network condition. However, the cost of solving the 0-1 ILP is not negligible in terms of time and memory even if it's on the server side, which is a NP-hard problem and no polynomial solution is available [22].

III. DESIGN RATIONALE

We illustrate the design principles of our code partition algorithm in this section. For a clear understanding, we first give a brief review of the call graph model which is commonly used in describing the calling relationship of an application. After that we explain why finding the best offloading and integrating points is a reasonable choice in a call graph.

A. Call graph model

The call graph is a directed weight graph where the vertex represents a method and the edge represents the calling relationship from the caller to the callee. Each vertex and each edge has two weights. The weights of the vertex are the cost of

executing the method on the cloud and on the mobile device. The weights of an edge are the cost of transferring the input and return states if the caller and callee methods are executed on different places. To get the application correctly run on two separate machines, the system needs to transfer all the potential states that could be accessed by the callee/caller. Specifically, the input state is the input parameter plus the public state which includes the current object's member variables (including nested member variables), the static classes, and the public static member variables. The return state is the return value plus this public state mentioned above.

For a given application, the execution cost is affected by input and device characters, while the transferring cost is related to the input and network condition. The metric of cost can either be time or energy consumption. While our partition algorithm is applicable to both metrics, we adopt time as the cost metric hereafter because the profiling process does not need any extra device (e.g., a power meter). Given that the computation capacity of cloud infrastructure is stronger than that of the mobile device, we assume that the execution time on the cloud is smaller than that on the mobile device without loss of generality. This assumption still holds when using energy consumption as the cost metric, where the energy cost on the mobile device is considered much more important than cloud.

It is worth mentioning that not all methods can be executed remotely [6, 8]. For example, methods that interact with the user interface should not be offloaded. Another type of methods that cannot be offloaded is the one that accesses I/O devices and hardwares of the mobile device, such as camera, GPS, accelerometer or other sensors. Finally, method that might cause security issues when executed on a different place should not be offloaded (such as e-commerce). The remoteabilibty of a method can be automatically detected using static analysis [8].

Fig. 2a is an illustration of the call graph of the face recognition application. The input object to be recognized is a 42KB (398×545) JPEG image, and the searching space is a set of 32 images with the same size. The yellow vertices are remoteable methods while the red ones are unremoteable due to I/O, hardware or external constrains. The number in the yellow vertex is the CPU cycles of that method in million cycles. The numbers on the edge are the sizes of the input and return states. We do not include system calls or external package method invocations. Only the methods developed by the application are considered here.

B. *Clustered offloading property*

Fig. 2a shows the partition result for the face recognition application using 0-1ILP when RTT is 150 ms. As can be seen from the figure, the offloaded methods are clustered together. Similar results are observed with different RTTs and applications. The clustered offloading partition layout implies that the offloading methods are likely to be executed sequentially, which is consistent with our intuition because it avoids the extra transferring cost of discrete offloads. Moreover, since the execution cost on the cloud is less compared to the execution cost on the mobile device, sequentially offloading further saves the cost by reducing the execution cost.

IV. Code Partition Algorithm

This section specifies the design of the code partition algorithm. First, we show that finding the best offloading and integrating points on a call graph is equal to finding them on the corresponding call link which is derived by depth-first search (DFS). Then we propose a linear time searching strategy to find the best offloading and integrating points on the call link. The overall complexity of the code partition algorithm (including the DFS conversion and the linear time searching) is $\Theta(V+E)$, where V is the number of remoteable methods and E is the number of invoking relations in the application.

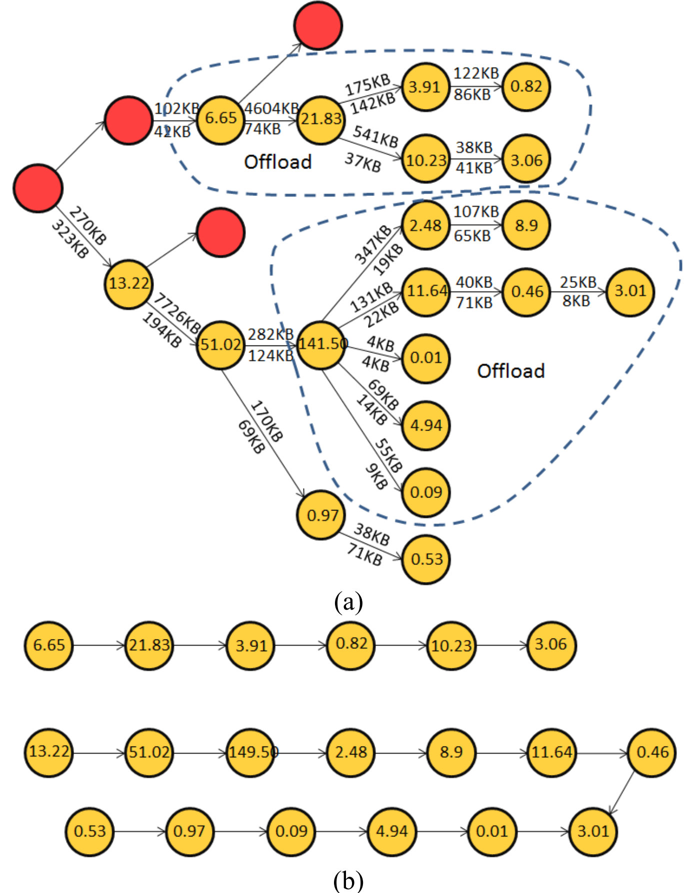

Figure 2. Call graph and call link for the face recognition application.

A. *Call link*

We have demonstrated that clustered offloading is ubiquitous on the call graph in section II. However, finding the best offloading and integrating points on the call graph is still an NP problem due to its tree structure, since every branching increases the searching space geometrically. The clustered offloading property itself does not alleviate the difficulty of finding a good partition in real time. We need a way to further reduce the solution space.

We observe that even though the call graph has the tree structure, where several callee methods could be called from one caller, they are executed sequentially within a single thread. This property still holds when the code is executed on separated places. From the time line of the execution in the

thread, the call graph can be converted into a call link that represents the executing order using DFS without changing the weights on the edge. Hence, finding the best offloading and integrating points on the call graph is essentially equal to finding them on the call link, and the optimal solution is the same. The searching complexity of the DFS algorithm is $\Theta(V+E)$.

For a single thread, the unremoteable methods separate the call link into several sub call links that does not interfere with each other. Only remotable methods are in a sub call link. Fig. 2b shows the sub call link of the face recognition application. Remoteable methods invoked by the same unremoteable method are considered to be in different sub call links. Fig. 3 is an example of two sub call links determined by one unremoteable method. Here, methods 2 and 3 belong to different call links. Since the executions of the sub call links are separated by the unremoteable method and thus they are not continuous in time, the best offloading and integrating points can be found on each the sub call link independently.

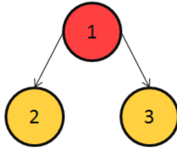

Figure 3. Sub call links separated by an unremoteable method.

B. *One time offload and optimal substructure*

The searching scheme is guided by two properties of the call link where the best offloading and integrating points are to be found. The one time offload property shows that there is only one pair of offloading and integrating points on each sub call link, while the optimal substructure property ensures that all methods share the same best integrating point.

1) *One time offload property*

The optimal number of the offloading and integrating points for the each sub call link is one.

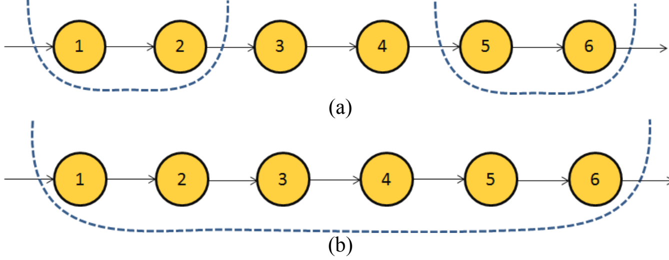

Figure 4. One time offload property.

Proof: for a chain of remoteable methods, if more than one pair of offloading and integrating points exist, then there must be methods that are embedded between the offloaded methods, like node 3 and 4 in Fig. 4. For this kind of methods, also offloading them is always a better choice than keeping them locally, since the cost of running a method remotely is always lower than running it locally and the transferring cost will be saved. Hence, one pair of offloading and integrating points is the optimal number for the each sub call link.

2) *Optimal substructure property*

For a given (sub) call link, every offloading points share the same optimal integrating point, and the performance improvement of offloading from method i is,

$$U_i = \begin{cases} \max\limits_{1\le k\le K}\left\{\sum_{j=1}^{k}\left(M_j - C_j\right) - I_1 - R_k\right\}, & \text{if } i=1 \\ U_{i-1} - \left(M_{i-1} - C_{i-1} - I_{i-1}\right) - I_i, & \text{otherwise} \end{cases} \quad (1)$$

where U_i is the utility or performance improvements when offloading starts from method i to its corresponding best integrating point, M_i and C_i are the costs of executing method i on the mobile device and the cloud respectively, I_{i-1} and R_{i-1} are the costs of transferring the input and return state for method i, and K is the number of methods on that (sub) call link.

Proof: for the first method in that (sub) call link, its corresponding optimal integrating point is found by traversing the entire link. $\sum_{j=1}^{k}(M_j - C_j)$ is the total performance gain by offloading from method 1 to k. I_1 and R_k are the transferring cost. For all the other methods except the first, their corresponding optimal integrating points have the optimal substructure. The utility of offloading from method i to any method is the utility of offloading from method i - 1 to the same integrating point minus the performance gain of having method i - 1 to execute on the cloud, which is $M_{i-1} - C_{i-1}$, plus the cost difference of transferring the input state of method i - 1 and method i. Since the rest of the utility function does not change with different integrating points, and the utility of integrating at method i 's best integrating point is the maximum among all the possible integrating points, method i shares the same best integrating point with method i - 1. Hence, all methods share the same best integrating point and have the optimal substructure property.

C. *Linear time searching*

Leveraging the properties of the best offloading and integrating points discussed in the two theorems, they can be easily found by a simple linear time searching scheme:

Algorithm 1: Linear time searching

$K \leftarrow$ *# of methods in the sub call link*
for $i \leftarrow 1$ *to* K
  (calculate performance gain of offloading from the 1st method to the ith)
  $gain_{1\,to\,i} \leftarrow \sum_{t=1}^{i}(mobile_t - cloud_t) - input_1 - return_i$
end
$U_1 \leftarrow \mathbf{max}\{gain_{1\,to\,i}, 1 \le i \le K\}$
integrating point $\leftarrow \arg\mathbf{max}\{gain_{1\,to\,i}, 1 \le i \le K\}$
for $j \leftarrow 2$ *to* K
  (calculate the utility of offloading from the jth method)
  $U_j \leftarrow U_{j-1} - (mobile_{j-1} - cloud_{j-1}) + input_{j-1} - input_j$
end
offloading point $\leftarrow \arg\mathbf{max}\{U_j, 1 \le j \le K\}$

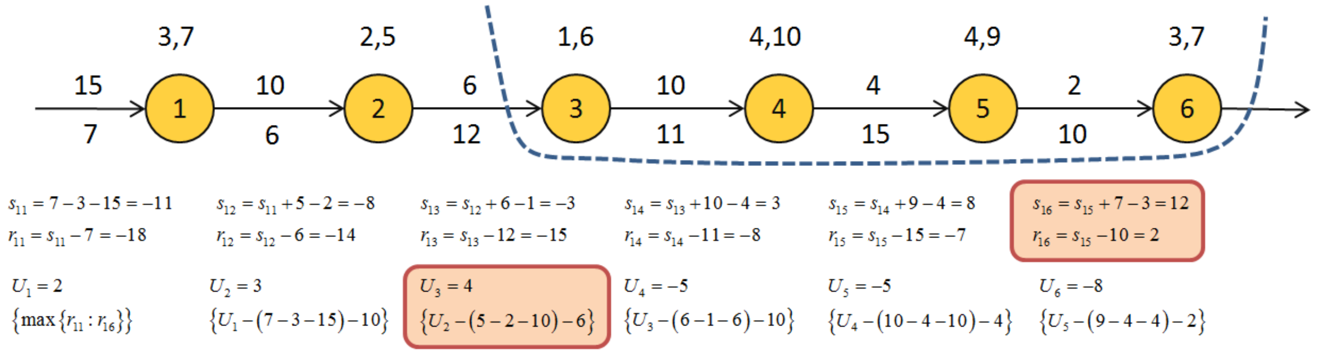

Figure 5. Example of Linear time searching scheme. s_{1i} is the time savings of offloading from method 1 to method i with only the cost of input state. r_{1i} is the net savings of offloading from method 1 to method i with the cost of both input and return state. U_i is the maximum savings of offloading from method i.

The pseudocode code in Algorithm 1 shows how to find the best offloading and integrating points in linear time. $mobile_i$ and $cloud_i$ are the cost of executing method i on the smartphone and on the cloud respectively. $input_i$ and $return_i$ are the cost of transferring the input and return state for method i. The first *for* loop finds the best integrating point by traversing the (sub) call link. The second loop finds the best offloading point. The process is also illustrated with a simple example in Fig. 5.

V. Evaluation

We implement our partition algorithm in C on a dual core Intel Pentium 2.0GHz processor, 4G RAM desktop. To get an understanding of the accuracy and speed of the call link approach, we also implement the 0-1 ILP solver for comparison by ourselves. The input of these algorithms is a linked list which represents the call graph with the following fields: execution time on the desktop, execution time on the mobile device, a pointer to its first child, a pointer to its next sibling, size of the input state to invoke this method, size of the return state, and the network bandwidth. The output of the two algorithms is the partition layout.

To obtain the execution time of a method, we insert a timer in each method and log the execution time both on the mobile device and the desktop. The smartphone being used here is the Samsung Galaxy ACE GT-S5830 with 278 MB RAM and Android 2.3.3 OS. The desktop is equipped with a dual core Intel Pentium 2.0GHz processor and a 4G RAM. To get the data size, we currently use the *Sizeof* tool developed by Roubtsov [13]. The *Sizeof* tool calculates an object's size in the heap, which is sufficient for obtaining the input of a partition algorithm. In real implementation scenario, we need actually access the object in heap. This could be done via the reflection attribute of the Java language.

We simulate the transferring time of the input and return state using the following rough estimation:

$$T = \frac{\text{state size}}{\text{BW}}. \tag{2}$$

As can be seen, we only consider transmission delay and ignore both the queuing delay and propagation delay in the estimation. The reason is that in real implementation, obtaining the bandwidth and queuing delay is nontrivial and energy costing, and the empirical way to get the transmission time is to use previous RTT times the data size ratio [6, 8]. Hence, as long as the estimation is proportional to data size, ignoring the other two delays would make real no difference if the results are evaluated in a proportional scale.

The face recognition application has 19 remoteable methods, while the natural language processing algorithm has 73 remoteable methods. The speed for both the 0-1ILP and our approach using call link is:

TABLE I. Execution time for Face Recognition and NLP

Time/ms	0-1 ILP	Call link
Face recognition	886.9383	4.5051
NLP	7282.8045	19.4476

As can be seen from the table, the execution speed of the call link approach is 2 orders of magnitude faster than the 0-1 ILP approach. The advantage is more distinct when the scale of the composable components grows. Note that the growing speed of the execution time of our solution is much slower than the 0-1 ILP approach, which shows the scalability of our call link approach. Here we only consider the execution time for the two partition algorithms, where the overall performance by applying these two algorithms is compared in Fig. 6 and 7.

Up to now, we only have the profiled data of these two applications. To get a broader scope of comparison of the execution speed with regard to method scale, we generate the call graphs from the three downloaded applications from Google Play. We use static analysis to obtain the calling relationship from the disassembled dex files. To get the remoteability of each method, we parse the disassembled dex files, detect standard API calls, and manually identify API calls that access hardware resources on the smartphone. Since we focus on the running speed, which is irrelevant to the profiling, we generate weights randomly for these three call graph.

The results of partition speed of these three applications are shown in Table II. The number in the parentheses is the amount of remoteable methods in that application. Even though the real calling relationship and remoteability might differ from what we obtained by static analysis, the result can give a general overview of the partitioning speed for off-the-shelf cognition

augmenting applications.

TABLE II. EXECUTION TIME FOR THREE DOWNLOADED APPS

Time/ms	0-1 ILP	Call link
Gesture Search (118)	125181.6469	32.8596
Adobe Photoshop Express (21)	1094.95	5.7810
Visidon AppLock (46)	3693.1344	12.0656

We define the partition accuracy as the number of correct decisions divided by the number of remoteable methods. We execute the call link and 0-1 ILP partition algorithms for several times with various RTTs and user input sizes. We emulate RTT for 100KB data in four conditions 50ms, 100ms, 150ms, and 200ms. For the face recognition application, we use the input object to be recognized is a 42KB (398×545) JPEG image, and use the same sized images as the searching space. We verify the searching space from 24 images to 32 images. For the NLP application, we verify the input text file sizes from 200 Byte to 2KB. Under these settings, the average partition accuracy is 92% for the face recognition application and is 91% for the NLP application.

Since code partition process is part of the whole mobile offloading process, we want to know the overall performance after applying the code partition algorithm. We define the overall performance as the sum of the time required for partition, one RTT for transferring the partition layout, and the execution and transferring time using that partition layout. Hence, the overall performance represents the total time required for the code offload process.

Fig. 6 and 7 illustrate the overall performance by using our partitioning algorithm for the face recognition application and NLP application. As can be seen from these figures, the total time of code offloading process increases when RTT grows longer for both the call link and 0-1 ILP approaches. When the RTT is smaller than 200ms (the average RTT to the first pingable point in 3G network), using the call link approach can save more time than the 0-1 ILP approach. Hence, the accuracy of our call link approach is good enough to provide an overall result.

VI. Conclusion and Discussion

In this paper, we present an efficient code partition algorithm for mobile code offloading with $\Theta(V+E)$ time. Instead of making an individual decision for each method like the 0-1 ILP model, our algorithm finds the offloading and integrating points for the whole program with all methods. Experimental results show that our algorithm achieves a speed improvement of 2 orders of magnitude over 0-1 ILP with more than 90% partition accuracy.

It is worth mentioning that our algorithm can serve as an enabling technique for the "Offloading-as-a-Service" scenario, where service providers offer commercial offloading service for mobile users. In that case, our partition algorithm can achieve low user-perceived latency while largely reduce the partitioning computation on cloud.

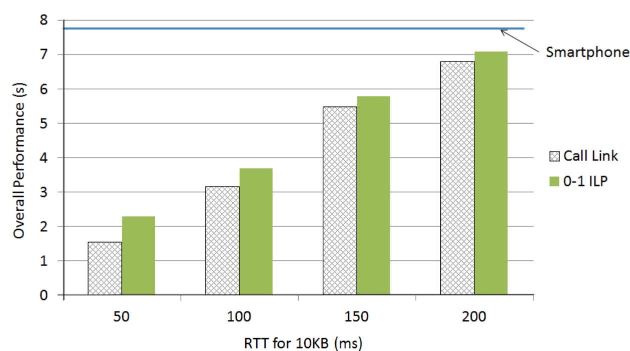

Figure 6. The overall performance for face recognition.

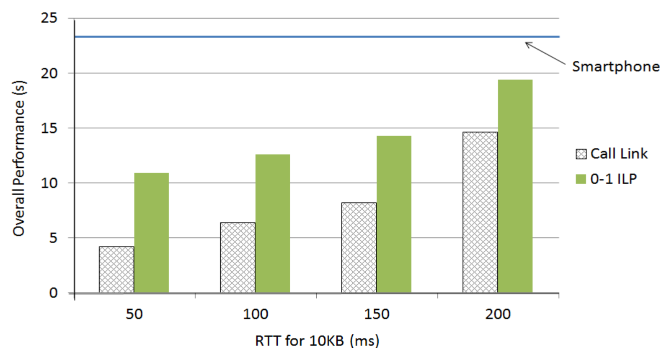

Figure 7. The overall performance for NLP.

References

[1] J. Cohen, “Embedded speech recognition applications in mobile phones: Status, trends, and challenges,” IEEE ICASSP 2008.

[2] “Apple Litigation Marks A New Era Of Expectations For Voice Recognition”, http://seekingalpha.com/article/432161-apple-litigation-marks-a-new-era-of-expectations-for-voice-recognition, March 2012.

[3] G. Hua, Y. Fu, M. Turk, M. Pollefeys, and Z. Zhang, “Introduction to the Special Issue on Mobile Vision,” International Journal of Computer Vision, vol. 3, pp. 277-279 , 2012.

[4] T. Park, J. Lee, I. Hwang, C. Yoo, L Nachman, and J. Song, “E-Gesture: a collaborative architecture for energy-efficient gesture recognition with hand-worn sensor and mobile devices,” SenSys 2011

[5] J. Huang, Q. Xu, B. Tiwana, Z. Morley Mao, M. Zhang, and V. Bahl, “Anatomizing application performance differences on smartphones,” MobiSys 2010.

[6] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making Smartphones Last Longer with Code Offload,” MobiSys 2010.

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

[8] B.-G. Chun and P. Maniatis. “CloneCloud: Elastic Execution between Mobile Device and Cloud”. EuroSys 2011.

[9] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai and David Wetherall, and Ramesh Govindan. “Odessa: Enabling Interactive Perception Applications on Mobile Devices”. MobiSys 2011.

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

[11] S. Osman, D. Subhraveti, G. Su, and J. Nieh. “The Design and Implementation of Zap: A System for Migrating Computing Environments”. OSDI 2002.

[12] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. "The Case for VM-based Cloudlets in Mobile Computing". IEEE Pervasive Computing, 8(4), 2009.

[13] Vladimir Roubtsov, Sizeof, http://www.javaworld.com/javaworld/javatips/jw-javatip130.html#resources

[14] J. Xia, "The third-generation-mobile (3G) policy and deployment in China: Current status, challenges, and prospects," Telecommunications Policy, 2011

[15] "Global mobile statistics 2012 Home: all the latest stats on mobile Web, apps, marketing, advertising, subscribers and trends", http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/

[16] "Global 3G and 4G Deployment Status," http://www.4gamericas.org/index.cfm?fuseaction=page&pageid=939

[17] Adobe Photoshop Express, https://play.google.com/store/apps/details?id=com.adobe.psmobile

[18] Gesture Search, https://play.google.com/store/apps/details?id=com.google.android.apps.gesturesearch

[19] Visidon AppLock, https://play.google.com/store/apps/details?id=visidon.AppLock

[20] http://code.google.com/p/dex2jar

[21] http://java.decompiler.free.fr/?q=jdgui

[22] http://en.wikipedia.org/wiki/Integer_programming