# Refactoring Android Java Code for On-Demand Computation Offloading

Ying Zhang[1,2] Gang Huang[1,2] * Xuanzhe Liu[1,2] Wei Zhang[1,2]
Hong Mei[1,2] Shunxiang Yang[1,2]

[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
[2]Institute of Software, School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, China

zhangying06@sei.pku.edu.cn; hg@pku.edu.cn; liuxzh@sei.pku.edu.cn; zhangwei11@sei.pku.edu.cn
meih@pku.edu.cn; yangsx07@sei.pku.edu.cn

## Abstract

Computation offloading is a promising way to improve the performance as well as reducing the battery power consumption of a smartphone application by executing some parts of the application on a remote server. Supporting such capability is not easy for smartphone application developers due to (1) correctness: some code, e.g., that for GPS, gravity, and other sensors, can run only on the smartphone so that developers have to identify which parts of the application cannot be offloaded; (2) effectiveness: the reduced execution time must be greater than the network delay caused by computation offloading so that developers need to calculate which parts are worth offloading; (3) adaptability: smartphone applications often face changes of user requirements and runtime environments so that developers need to implement the adaptation on offloading. More importantly, considering the large number of today's smartphone applications, solutions applicable for legacy applications will be much more valuable. In this paper, we present a tool, named DPartner, that automatically refactors Android applications to be the ones with computation offloading capability. For a given Android application, DPartner first analyzes its bytecode for discovering the parts worth offloading, then rewrites the bytecode to implement a special program structure supporting on-demand offloading, and finally generates two artifacts to be deployed onto an Android phone and the server, respectively. We evaluated DPartner on three real-world Android applications, demonstrating the reduction of execution time by 46%-97% and battery power consumption by 27%-83%.

***Categories and Subject Descriptors*** D.2.3 [*Software Engineering*]: Coding Tools and Techniques – Object-oriented programming; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement – Enhancement; C.2.4 [*Computer-Communication Networks*]: Distributed Systems – Client/server, Distributed applications

***General Terms*** Experimentation, Languages, Performance

***Keywords*** computation offloading; bytecode refactoring; energy; Android

## 1. Introduction

Android [1] is an open source mobile platform for smartphones, and it has gained more than 59% of smartphone market share in the first quarter of 2012 [2]. Hundreds of thousands of developers have produced more than 490 thousands Android apps (a special kind of Java applications, app is short for application) since the platform was first released by Google in 2008 [3] [4]. Following the fast improvement of smartphone hardware and increased user experience, Android apps try to provide more and more functionality and then they inevitably become so complex as to make the two most critical limits of smartphones worse.

The first limit is the battery power. Complex Android apps usually have intensive computations and consume a great deal of energy. For instance, the top 10 downloaded Android apps such as *Fruit Ninja* and *Angry Birds* on the Google Play [3] (formerly known as the Android Market) are all the complex ones that can drain the battery power in about 30 minutes if running on HTC G13 [5]. Although the battery capacity keeps growing continuously, it still cannot keep pace with the growing requirements of Android apps [6].

The second limit is the diversity of hardware configurations, which gives smartphone users very different experiences even running the same app. For example, among the

* Corresponding author



*OOPSLA'12,* October 19–26, 2012, Tucson, Arizona, USA.

best-selling HTC smartphones in 2011, G13 has a 600 MHz CPU, G11 and G12 have a 1GHz CPU, and G14 has a 1.2 GHz CPU. The same *Fruit Ninja* app runs very slowly on G13, faster on G11 and G12, and the fastest on G14. Generally speaking, the lower hardware configuration of a phone implies the lower performance of the apps running on the phone. Users usually stop using the *slow* apps even if the root cause of the poor user experience is due to the low hardware configuration of the phone. Such factor can lead to a big loss of the app market for developers and vendors.

Computation offloading is a popular technique to help improve the performance of an Android-like smartphone app and meanwhile reduce its power consumption. Offloading, also referred to as *remote execution*, is to have some computation intensive code of an app executed on a nearby server (the so-called surrogate, e.g., a PC), so that the app can take advantage of the powerful hardware and the sufficient power supply of the server for increasing its responsiveness and decreasing its battery power consumption.

Computation offloading is usually implemented by a special program structure, or a design pattern, that enables a piece of code to execute locally or remotely and handles the interactions between the local and the remote code without impact on the correctness of the functionality. With respect to features of smartphones, three advanced issues have to be dealt with:

- Correctness: some code, e.g., that for GPS, gravity, and other sensors, can run only on the smartphone. Therefore developers have to identify which parts of the app cannot be offloaded;
- Effectiveness: the reduced execution time must be greater than the network delay caused by computation offloading. Therefore, developers have to calculate which parts are worth offloading;
- Adaptability: one of the most important natures of mobile computing is the diverse, frequent, and rapid changes of user requirements and runtime environments, which may lead to changes of computation offloading. For example, if the remote server becomes unavailable due to unstable network connection, the computation executed on the server should come back to the smartphone or go to another available server on the fly. Developers have to consider such changes in computation offloading.

The preceding issues are not easy to deal with and, more importantly in practice, a solution should be applied to legacy apps in a cost effective way considering the huge number of today's Android apps.

In this paper, we present an automatic approach to realizing computation offloading of an Android app. Our paper makes three major contributions:

- A well-designed pattern to enable an Android app to be computation offloaded on-demand. All Java classes of the app are able to interact with each other locally or remotely. The computations of a class can be offloaded dynamically and only the interactions between the smartphone and the server go through the network stack.
- A refactoring tool, named DPartner, to automatically transform the bytecode of an Android app to implement the proposed design pattern. DPartner first analyzes the bytecode for discovering the parts of an app that are worth offloading, then rewrites the bytecode to implement the design pattern, and finally generates two artifacts to be deployed onto the Android smartphone and the server, respectively. Refactoring is transparent to app developers and supports legacy apps without source code.
- A thorough evaluation on three real-world Android apps. The evaluation results show that our approach is effective. The offloaded apps execute much faster (reducing execution time by 46%-97%) and consume much less battery energy (reducing power consumption by 27%-83%) than the original ones.

We organize the rest of this paper as follows. Section 2 presents the design pattern for computation offloading. Section 3 gives the implementation of DPartner. Section 4 reports the evaluation on Android apps. Section 5 discusses related work and we conclude the paper in Section 6.

## 2. Design Pattern for Computation Offloading

An Android app is a Java program, whose building blocks are classes. Any meaningful computation is implemented as a method of a class, which uses the data and methods internal of the same class or invokes some methods of other classes. As a result, offloading computations can be implemented as remotely deploying and invoking a single class or a set of classes performing the computation. The goal of our approach is to automatically refactor Android apps into the ones implementing the design pattern for such deployment and invocation.

The principle of refactoring is to restructure the given code without altering the external functionality [9]. Generally speaking, refactoring is characterized by three aspects [9]: (1) the structure of the original code, i.e., source structure; (2) the structure of the target code, i.e., target structure; (3) a sequence of code refactoring steps that transform the original code to the functionally equivalent target code, so that the target code are finally able to take on the desired program structure. Thus, we introduce these aspects one by one in detail as follows.

### 2.1 The Source Structure and the Target Structure

In Java, the object reference determines whether two classes interact with each other locally or remotely. The source structure of the given code in any standalone Android app all take on the "local reference" program structure, i.e., the so-called "in-VM reference" program structure as shown in


**Figure 1.** Local invocation (source structure)

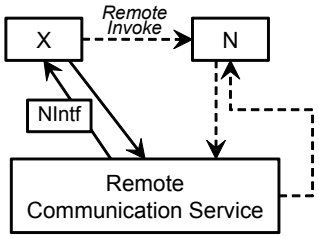


**Figure 2.** Remote invocation (source structure)

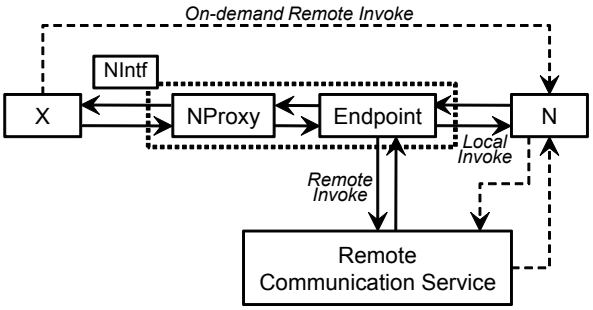


**Figure 3.** On-Demand remote invocation (target structure)

Figure 1. Class *X* first gets the in-VM reference of class *N* and then invokes the methods of *N*. Such a structure does not support offloading any computation in class *N* because if *N* is offloaded to a remote server, *N*'s in-VM reference held by *X* becomes invalid, i.e., it cannot help the VM pass the method invocations to *N*, and *X* will fail to obtain the new in-VM reference of *N*.

Figure 2 presents the typical code structure that enables class *X* to invoke the methods of a remote class *N*. That is, *X* gets a remote reference to *N* from a remote communication service, then uses the reference to interact with *N* remotely. The remote communication service is responsible to associate *N*'s reference with *N* across the network. Take Android AIDL [39] as an example, its *ServiceConnection* can be seen as the main part of the remote communication service. *X* gets a reference to *N* in the *onServiceConnected* method of *ServiceConnection*, then uses the reference to invoke the methods of *N*. Such a structure can have the computations in *N* offloaded to a remote server, but suffer serious performance penalty if *N* and *X* are both in the same VM. As mentioned above, whether *N* is offloaded or not is determined dynamically and may change time to time. If *N* is not offloaded, all the interactions between *X* and *N* in such a structure will still go through the time-consuming network stack, which is contrary to the goal of computation offloading, i.e., improving performance and save energy. We have done an experiment on the standard Android SDK sample: net.learn2develop.Services [40]. *X* (*MainActivity*) uses the *ServiceConnection* to get the reference to *N* (*MyService*). After that it uses the reference to call *N*'s methods. Compared with local interactions, such *remote interactions* in the same VM increase execution time by 680% and battery power consumption by 287%.

Figure 3 presents the target structure we propose for on-demand computation offloading, i.e., allowing *X* to effectively invoke *N* no matter the two are running in the same VM or in different VMs across the network. The core of the structure is composed of two elements: *proxy* and *endpoint*.

A proxy, *NProxy* in Figure 3, acts the same as the proxied class *N* except that it does not do any computation itself, but forwards the method invocations to the latter. If the location of the proxied class *N* is changed from local to remote, or from one remote server to another, *NProxy* keeps unchanged so that the caller, class *X*, will not get noticed.

The endpoint is responsible for determining the current location of *N* and for the truly crossing network communication from *X* to *N*. When *N* is running in a remote VM, the endpoint will take advantage of a given remote communication service to get a reference to *N* and pass it back to *X*, then *X* can use the reference to invoke *N* remotely. When *X* and *N* both run in the same VM, the endpoint will directly obtain the in-VM reference of *N*, so that *X* can invoke *N* without going through the network stack. The endpoint decouples the caller and the remote communication service. A smartphone can dynamically change its network connection with the same server among multiple protocols, e.g., Wi-Fi, and 3G. The endpoint automatically adapts to such changes so that the interacted app classes are unaware of them.

### 2.2 Refactoring Steps Overview

A sequence of refactoring steps will be performed on the Java bytecode of a given standalone Android app, so that the source structure of the code shown in Figures 1 and 2 can be transformed to the target structure shown in Figure 3. We implement a tool, called DPartner [8], to automatically execute the refactoring steps, as shown in Figure 4.

*Step 1: detect which classes are movable*. For a given app, DPartner automatically classifies the Java classes (i.e., bytecode files) into two categories: Anchored and Movable. The anchored classes must stay on the smartphone because they directly use some special resources available only on the phone, e.g., the GUI (Graphic User Interface) displaying, the gravity sensor, the acceleration sensor, and other sensors. If being offloaded to the server, these anchored classes cannot work because the required resources become unavailable. Besides those anchored classes, all other classes are movable, i.e., can execute either on the phone or on the server.

*Step 2: make movable classes be able to offload*. When a class is offloaded, the local invocation structure between this class and its interacted classes should be transformed to the on-demand remote invocation structure, e.g., generating the proxy of the callee class and rewriting the caller class to

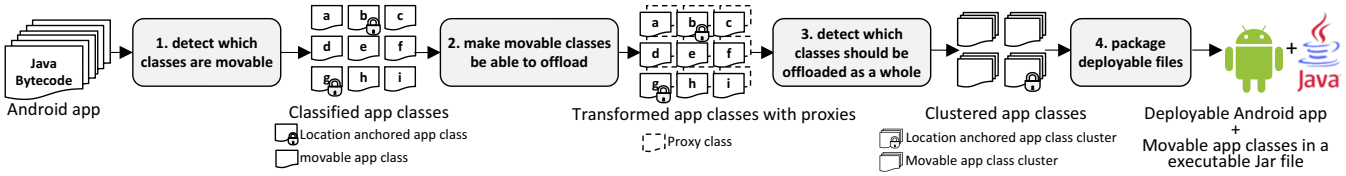


**Figure 4.** Refactoring steps

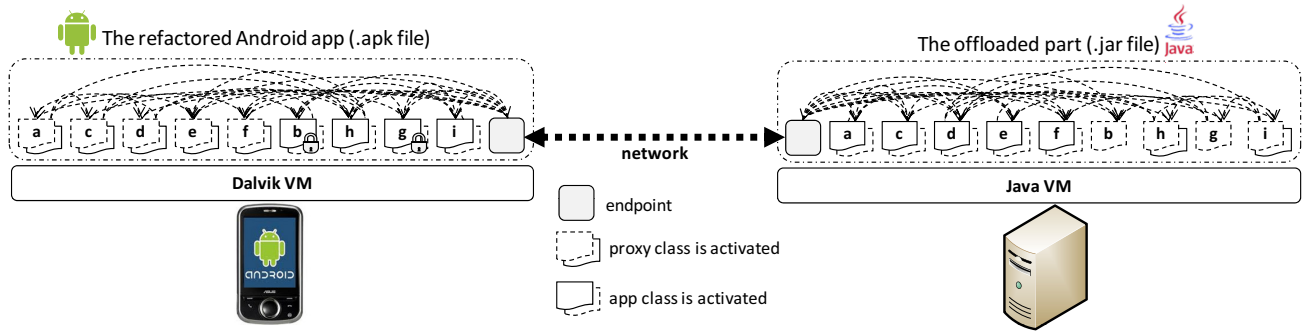


**Figure 5.** The runtime architecture of an offloaded Android app

equip the proxy. Note that, if an anchored class is invoked by an offloaded class, the latter needs the former's proxy. Since which movable classes are offloaded is determined at runtime, we have to generate the proxies for all callee classes and rewrite the corresponding caller classes except that the caller and callee are both anchored. In our approach, a proxy class will be made to act exactly like its proxied app class. That is to say, these two classes will extend the same inheritance chain, implement the same interfaces, have the same method signatures, etc. In this way, class *X* will feel no difference when using the proxy of class *N* instead of *N* itself. The specific features of Java have to be considered when generating the proxies, which include the static methods, final methods/classes, public fields, inner classes, arrays, etc. The details of proxy generation and class transformation will be presented in Section 3.

*Step 3: detect which classes should be offloaded as a whole*. There are numerous rules and algorithms to determine which movable classes should be offloaded [7]. As mentioned before, such a decision has to be made and changed at runtime due to the mobility of smartphones. Meanwhile, making decision at runtime inevitably consumes resources and therefore it is valuable to have done some pre-processing to simplify runtime decision. DPartner employs a well-known rationale for pre-processing based on class clustering [24], i.e., the frequently interacted classes should be offloaded as a whole. In this way, it cannot only avoid the time-consuming network communication between these classes, but also help accelerate runtime decision. For instance, if class *X* and class *N* interact with each other frequently, the endpoint only needs to profile the execution trace of *X* to determine offloading at runtime. When *X* is about to offload, *N* will then be profiled to decide whether it should be offloaded together. It is unnecessary to continuous profile *N* for reducing runtime overhead.

*Step 4: package deployable files*. The input of DPartner is the Java bytecode files of an Android app as well as the referenced resource files, e.g., images, xml files, and jar libs. After going through the above three steps, DPartner will package the files and then generate two artifacts. The first is the refactored Android app, i.e., an .apk file, which is ready to be installed on a phone. The second is an executable jar file, which contains the movable Java bytecode files cloned from the refactored app. Both artifacts include the code of the endpoint and the communication service.

### 2.3 An Illustrative Example

Figure 5 shows an example of the runtime architecture of an offloaded Android app. This app is composed of six classes, with their names from *a* to *i*, respectively. In *Step 1*, DPartner finds that class *b* and *g* are both anchored classes, so that they should always run on the phone. All the other classes are movable. In *Step 2*, each app class is transformed and gotten its corresponding proxy class. In *Step 3*, DPartner finds that class *a*, *c*, *d*, *e*, and *f* are closely related to each other, and then clusters them for offloading as a whole. In *Step 4*, DPartner packages all the movable classes, the proxies, and the endpoint classes in to a jar file that will be deployed and executed on the server, and packages all the app classes, the proxies, and the endpoint classes as an Android apk file that will be deployed and executed on the phone.

At runtime, the endpoint predicts that offloading class *d* can improve the whole app performance, and then deactivates the *d* running on the phone and activates the *d* deployed on the server. The *d*'s proxy on the phone will forward the incoming method invocations to the *d* on the server. Since

class *d* is clustered with class *a*, *c*, *e*, and *f* in *Step 3*, these classes will also be offloaded to the server when necessary by going through the deactivating, state synchronizing and activating procedures as class *d*. Any invocations to these app classes will be forwarded to the server by the endpoint. When the offloading-related conditions become unfavorable, e.g., the phone gets far away from the server which leads to a high network latency, the computations of *a*, *c*, *d*, *e*, and *f* can get back to the phone by automatically deactivating them on the server and reactivating them on the phone. All invocations to these classes will be redirected to the phone by the endpoint.

# 3. Implementation of DPartner

## 3.1 Detect Movable Classes

DPartner classifies a Java class into anchored or movable via bytecode analysis. In an Android app, an anchored class must have one of the following features: (1) there are the "native" keyword existing in the class methods; (2) the class extends/implements/uses the Android system classes that are already regarded to be anchored by DPartner. For instance, the "android.view.View" class is used as the parent class for drawing the GUI of an Android app, so that it is classified to be the anchored class by DPartner. If a class is found to extend this class, it will also be anchored by DPartner. For another instance, the "android.hardware.*" classes are responsible for handling the camera and sensors of an Android smartphone. Therefore, they and the classes using them will be automatically anchored by DPartner. The rest of the classes will be classified as movable classes.

The above classification procedure may cause false positives and false negatives. A *false positive* means that a class is classified as a movable one, but the developer is unwilling to do this. For instance, the developer does not want the "cn.edu.pku.password" class be offloaded due to privacy concerns, although this class is classified as a movable one. A *false negative* means that a class is classified as an anchored one although it can actually be offloaded. For instance, as there are "native" keywords in its methods, the "cn.edu.pku.nativeCall" class will be classified as anchored automatically. However, such methods may never be invoked, so that this class can be offloaded. By analyzing the bytecode of an app, DPartner is able to tell whether an anchored class is invoked by the other classes of the same app. However, it cannot tell whether the class (acting as an Android service [38]) will be used by other apps, which may lead to false negatives of classification.

Therefore, we can see that DPartner takes a conservative approach to detecting movable classes, which can at least guarantee that a refactored app work correctly no matter being offloaded or not. To refine the classification results of anchored classes, DPartner provides a configuration file in which the naming patterns of anchored classes such as "android.hardware.usb.*" can be listed. In fact, the automatic classification procedure is carried out based on a predefined naming pattern list that takes effect just like the configuration file. At present, we have predefined 63 and 72 naming patterns for Android 2.1 and 2.2, respectively. Compared with Android 2.1, Android 2.2 has 96% naming patterns unchanged, 1% deleted, 1% revised, and 2% added. Developers can edit the configuration file to make the classification step adapt to the specific structures of each unique Android app and the platform. Of course, DPartner can force a movable class to be anchored by indicating it in the configuration file. For instance, developers can write "cn.edu.pku.password" in the file so that DPartner will automatically classify this class to be an anchored one.

## 3.2 Generate Proxies

One of the biggest challenge when generating proxies is to make a proxy act exactly like its proxied class. For instance, the app class *X* invokes the methods of another app class *N*. In the original code of *X*, a casting operation is done on *N* to *N*'s parent *NParent*. If we just change *N* to *NProxy* in *X*'s code, but not let *NProxy* extend class *NParent*, the casting operation will fail. Therefore, the proxy generated by DPartner will have the same program structure as the proxied class. Especially, the proxies themselves will maintain the same hierarchical structure as the proxied classes. For instance, *N* extends *NParent*, *NProxy* should also extend *NParent*'s proxy, so that any invocations to the inherited methods and constructors in *N* from *NParent*, can be forwarded first from *NProxy* to *NParent*'s proxy, and finally to *NParent*.

Java interface is used to separate the external representation and the internal implementation of a class. From the view of an interface, the proxy class and the proxied class are the same if they implement the same interface. Therefore, DPartner will automatically extract the interfaces to represent an app class and its proxy. For instance, there is an app class *N*, DPartner will (1) extract all the method signatures of *N* to form an *NIntf* interface; (2) make *NIntf* "extend" the *NParentIntf* interface of *N*'s parent *NParent*, so as to maintain the inheritance hierarchy; (3) make *N* implement *NIntf*; and (4) make *NProxy* also implement *NIntf*. After that, DPartner will rewrite all the other classes (e.g., the class *X*) from using *N* to *NIntf*. However, to the static methods of *N*, as they are not allowed in Java interfaces, DPartner will directly use the corresponding static methods of *NProxy* to forward the method invocations.

## 3.3 Transform App Classes

The app classes should also be rewritten to adapt to the offloading requirements. Given an app class, DPartner will automatically transform it using the following transformers:

1. *Field2Method transformer*: The non-private fields of a class, i.e., with the *public*, *protected*, or *null* modifier, can be used by other classes outside of its class scope. However, if the class is offloaded, its callers can never get the reference to these fields. To solve this problem, the transformer will au-

```
public class Integer_Array_Dimension1 implements
                Integer_Array_Dimension1_Intf{
   private Integer[] _array;
   public Integer aaload(int position){
         return _array[position];
   }
   public void aastore(int position, Integer elem){
               _array[position] = elem;
   }
   ...
}
```

**Figure 6.** The treatment to (Integer) array type

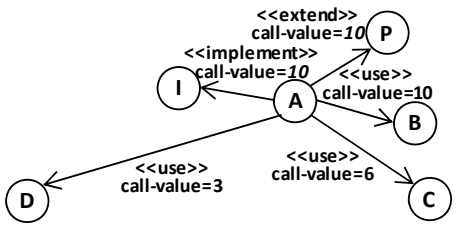


**Figure 7.** The classes relationships in a call graph

tomatically generate the public getter/setter methods for the non-private fields of a given class *X*, and then change their modifier to *private*. After that, the transformer will change all the classes that get/set the fields of *X* to use the corresponding getter/setter methods. In addition, to help state synchronization, the transformer also generates the getter/setter methods for the *private* fields. In this way, the inner states of a class instance can be collected and injected to enable state synchronization when the computations of a class instance are offloaded on-demand.

2. *Array transformer*: Given a class *N* that has an array type field being used by another class *X*, if *X* changes the element value of the array, *N* will see the changed value. That is because, in Java, an array is passed by reference. However, if *X* and *N* are running in different VMs, the array has to be passed by value between *X* and *N*, so that it can be transferred over the network. Under such a circumstance, *X* and *N* each have a copy of the array. If *X* changes the value of its copied array, but does not pass the array back, *N* will still use the old array, which makes the array value inconsistent. Therefore, the pass-by-reference feature of arrays should be kept. The array transformer wraps an array by using a functional equivalent class shown in Figure 6, which encapsulates the get/set operations of an array.

For example, class *N* has a field *intArray* with the Integer array type. Class *X* changes the value of *intArray* as the following code:

```
n.intArray[5]=3;
```

Then the above code will be replaced by the following code:

```
Integer_Array_Dimension1_Intf arrayIntf =
     n.getInteger_Array_Dimension1();
arrayIntf.aastore(5, new Integer(3));
//where n is an NProxy object represented by NIntf.
//arrayIntf is an Integer_Array_Dimension1_Proxy
  object represented by Integer_Array_Dimension1_Intf.
```

In this way, the get/set operations in *X* on the *intArray* will be intercepted by the corresponding proxy class, and finally be forwarded to *N*. Otherwise, these operations will be done directly by using the aastore/aaload bytecode instructions that cannot be intercepted, which will lead to the value inconsistency of *intArray* in *X* and *N*. Transforming multi dimensional array is similar to single dimensional array. For example, Integer[][] will be wrapped as an Integer_Array_Dimension2 object. The type of the "_array" field will be changed to Integer_Array_Dimension1. The get/set operations on the one-dimensional elements will also be changed to the aaload/aastore methods of the Integer_Array_Dimension1.

3. *ServerObject transformer*: To make sure an invocation received by a proxy is correctly forwarded to the corresponding proxied class instance, this transformer will make an app class implement a specific *ServerObject* interface, which has two methods: *getID* and *getProxyForSerializing*. The *getID* method will return an Integer value to identify the app class instance, which is used to correlate this object with its proxy. The *getProxyForSerializing* method is used to handle the object serialization issue in the callback style method invocation described in Section 3.6.

4. *Other transformers*: They handle the specific features of classes, including the anonymous constructor, inner class, "abstract" keyword, "final" keyword, etc. Their details are published on the website of DPartner [8].

### 3.4 Cluster App Classes

To improve the performance of runtime decision about which movable classes to offload, DPartner will cluster the frequently interacted classes and to offload them as a whole. It treats each app class as a node in the call-graph [26] of an Android app. The edge between two nodes means that the classes have one of the three relationships: *extend*, *implement*, and *use*. As shown in Figure 7, class *A* extends *P*, implements *I*, and uses class *B*, *C*, and *D*. The call-value, i.e., edge weight, from *A* to *B* is ten, which means that by static bytecode analysis, it is found that *A* calls the methods of *B* ten times in its code. The call-value from *A* to *C* is six, which is less than that of *A* to *B*. It denotes that *A* depends on *B* more than on *C*. Class *P* is the parent of *A*, however, by code analysis, the method call from *A* to *P* may be less than that of *A* to *B*. In order to highlight the importance of *P*, DPartner will calculate the maximum call-value from *A* to other nodes, then assign this value to the edge from *A* to *P*. Thus in the following clustering procedure, *A* and its parent *P* will more likely be clustered together. DPartner will do the same thing on the edge of *A* to *I*.

DPartner treats the above call graph as $G$=($V$, $E$), where $V$ is the set of nodes in the graph, and $E$ is the set of edges. A cluster is defined as a sub-graph $G'$ = ($V'$, $E'$), where

$V' \subseteq V$, and $E' \subseteq E$. The nodes in $V'$ has a much higher call-value with each other than with the nodes in $V$ - $V'$. DPartner employs the Girvan and Newman [22] (shorten as G-N), a classical and often used clustering algorithm, to reveal the clusters in a graph. In the G-N algorithm, the "betweenness" of an edge is the number of shortest paths between all pairs of nodes that pass through the edge in graph *G*. The sum of the call-value of the edges on the shortest path between two nodes is smaller than that of all the other paths connecting the same nodes. The edges lying between clusters are expected to be those with the highest *betweenness* and the lowest call-value. Thus by removing such edges recursively, G-N algorithm can find a candidate cluster-separation of a graph.

DPartner uses static analysis to calculate the call-values. Comparing with dynamic analysis, the static one does not need extra auxiliary inputs such as test cases, thus can be carried out automatically. However, the call-value calculated by static code analysis may be imprecise because the number of method-calls lying between the *loop* and the *jump*-like instructions cannot be calculated precisely unless the code is actually executed. For instance, the method call from *A* to *D* in Figure 7 is put in a while loop. By static code analysis, it is found that *A* calls *D* 3 times. When clustering, *A* and *C* will more likely be clustered together because they have a higher call-value than *A* to *D*. However, when the app is running, *A* will call *D* 3n times (n is the loop number) . If *A* and *D* are not clustered together and are offloaded into different VMs, their in-between network communication will be very high.

Fortunately, some research projects show that the closely related classes often have highly similarity in semantics [25]. Thus, DPartner leverages the semantic similarity information to modify the call-value between each two classes for making the value more accurate. DPartner analyzes the names and contents of each two class files to decide their semantic similarity. It will segment the class name and the textual contents of a class file into a term vector: T = $<t_1, t_2, t_3, ..., t_m>$. The textual contents include the names and types of methods, local variables, etc. For simplicity, we use the class name as an example. There are two classes, $class_1$ with $name_1$: "insa.android.andgoid.strategy.Pattern", $class_2$ with $name_2$: "insa.android.andgoid.strategy.PatternStone". $name_1$ will be segmented as <insa, android, andgoid, strategy, pattern>; while $name_2$ will be segmented as <insa, android, andgoid, strategy, pattern, stone>. Then by using the Jaccard Coefficient, a classic term-vector based text matching algorithm [23], we can tell whether $name_1$ and $name_2$ have a high similarity in texts, which indicates the corresponding classes may be highly related. To make it more clear, the semantic similarity of these two classes is calculated as follows:

$$SemSim(class_i, class_j) = JacSim(T_i, T_j) = \frac{N_{ij}}{N_i + N_j + N_{ij}}$$

**Table 1.** Class Clustering

```
1.  c = inputApplicationClasses()
2.  g = buildCallGraph(c)
3.  callValue = computeCallValue(g)
4.  semSim = computeSemSim(c)
5.  callValue = updateCallValue(callValue, semSim)
6.  b_set = computeBetweenness(g, callValue)
7.  while (clusters_number < threshold)
8.        b_maxSet = maxSet(b_set)
9.        e_removedSet = removeEdges(b_maxSet)
10.       g = g - e_removedSet
11.       b_set = computeBetweenness(g, callValue)
12. clusters = disconnectedSubgraphs()
```

In the above equation, $N_{ij}$ represents the total number of terms contained in both $T_i$ and $T_j$. For instance, the term "pattern" appears in both $name_1$ and $name_2$, so it will be counted in $N_{12}$. The term-containing check will be performed at the etyma level. That is to say, if "patterns" appears in $T_i$, and "patterned" appears in $T_j$, these two will be considered as the same one. $N_i$ represents the total number of terms contained in $T_i$ but not in $T_j$. $N_j$ represents the total number of terms contained in $T_j$ but not in $T_i$. DPartner will add such a *SemSim* value to the call-value of each corresponding edge in an app's call-graph as below, so as to use the G-N algorithm to reveal the class clusters. The new call-value is calculated as follows:

$$CallValue_{A\to B} = \alpha \times \frac{CallValue_{A\to B}}{\max(CallValue_{A\to X})} + \beta \times \frac{SemSim_{A\to B}}{\max(SemSim_{A\to X})},$$
$$A,B,X \in V, \text{ and } A \to X \in E; \quad \alpha,\beta \in (0,1), \text{ and } \alpha + \beta = 1$$

The whole clustering procedure is shown in Table 1. The ranges of the *threshold* are [1, $N_{class\_number}$], where $N_{class\_number}$ is the number of classes in a given Android app. Therefore, the extreme cases of clustering are: (1) all the classes belong to a single cluster; (2) every single class belongs to its single class cluster. DPartner will iterate the values in [1, $N_{class\_number}$] to get the clustering hierarchy graph of a given Android app as shown in Figure 8.

For example, when *threshold* is 2, the given app classes are clustered into two clusters. One cluster is composed of class *a*, *c*, *d*, *e* and *f*. The other is composed of *b*, *g*, *h* and *i*. As the second cluster contains anchored classes, i.e., *b* and *g*, thus this cluster is considered as an anchored class cluster, which means that the classes in such a cluster should never be offloaded together to execute on the server.

DPartner will store the above clustering information into the refactored Android app. Such information will be leveraged by the endpoint to decide which classes should be offloaded as a whole at runtime. To reduce the overhead of runtime decision, the endpoint leverages a selective monitoring strategy, i.e., it monitors only the execution trace of some of the most computation-intensive movable classes. If one monitored classes is predicted to be suitable for offloading, the other classes that are closely related to it will also be

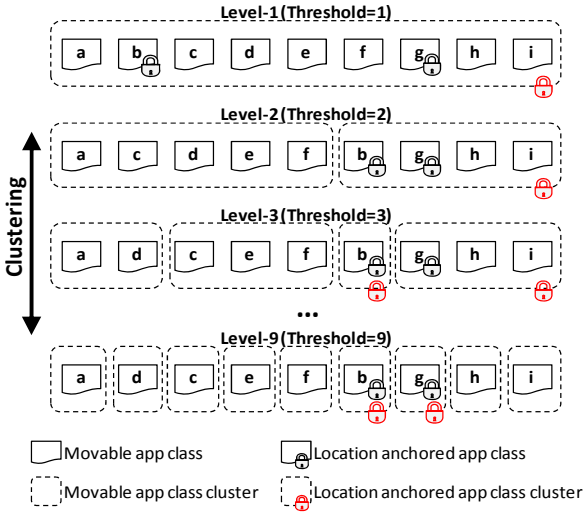


**Figure 8.** The clustering hierarchy graph

offloaded when necessary. For example, the most computation intensive class of an Android app is *a*. If the endpoint predicts that *a* should be offloaded, then it will search the clustering hierarchy graph from the top down to find the first movable cluster that contains *a* and meanwhile with all its classes being suitable for offloading. At first, the endpoint examines Level-1 in Figure 8, and finds that the cluster at Level-1 is anchored. Therefore, the endpoint goes on to examine Level-2. The movable cluster at Level-2, i.e., the cluster $\{a, c, d, e, f\}$, happens to contain *a*. At this time, the endpoint will use the prediction algorithm presented in the next section to check whether the classes in the cluster are all suitable for offloading. Otherwise, it will go on to search the next level to find a cluster that can be offloaded as a whole. The extreme case of the end of searching is at Level-9, where the endpoint will surely find a cluster being suitable for offloading, i.e., the single class cluster with only class *a*.

### 3.5 Determine the Computations to be Offloaded

As shown in [27], a class that has more bytecode instructions is usually the one that costs more computing resources to run, e.g., CPU and memory. Therefore, DPartner calculates the instruction numbers of each movable app class to find the top *n* (e.g., *n*=3) computation intensive classes. Then it will store these information into the refactored app. The endpoint will monitor only these top *n* classes to decide whether offloading is necessary by executing the following prediction algorithms at regular intervals (e.g., 1 second). The monitoring of the other classes is carried out as required, which helps reduce the runtime overhead on the smartphone.

A class instance must satisfy both the following two formulas when it is predicted to be suitable for offloading:

(a) $\forall$method *m* of class *c*, $t_{m\_phone}/t_{m\_offload} = t_{m\_phone}/(d_{m\_input}/r + t_{m\_phone}/i + d_{m\_output}/r) \geq \alpha$, where $\alpha \geq 1.5$. $t_{m\_phone}$ is the execution time when *m* is running on the phone, while $t_{m\_offload}$ is the total execution time when

```
public class NProxy extends NParentProxy
                     implements NIntf {
   public SIntf methodM(SIntf s, TIntf t){
      return (SIntf)
         ProxyFacade.getEndpoint().invokeMethod(
             "foo.bar.N", //offloaded class
             this.serverobjID,//class instance ID
             //bytecode-level method signature
             "methodM(Lfoo/bar/intf/SIntf;➥
             Lfoo/bar/intf/TIntf;)➥
             Lfoo/bar/intf/SIntf;",
             new Object[] {s, t} //parameters
         );
   }//end methodM
}//end NProxy
```

**Figure 9.** The method forwarding chain from a proxy to the endpoint

*m* is executed on the server; $d_{m\_input}$ is the method's input parameters in bytes; $d_{m\_output}$ is the method's return values in bytes; *r* is the data transmission rate over the network; $i = cpu_{server}/cpu_{phone}$ is the CPU cycle ratio between the server and the phone.

(b) $(E_{cpu\_offload} + E_{wifi\_or\_3G}) \leq E_{cpu\_local}$. *E* is the Android app's power consumption per time unit, e.g., 1 second. The inequality means that if class instance *c* is offloaded, the app's power consumption should never be greater than that when *c* is running on the phone. $E_{wifi\_or\_3G}$ can be calculated by monitoring the Wi-Fi/3G config files of the phone [29] [30]. $E_{cpu\_offload}$ and $E_{cpu\_local}$ can be calculated by using the mapping algorithm between the bytecode instructions and energy consumption proposed in [27].

### 3.6 Offload Computations at Runtime

As shown in Figure 5, its up to the endpoint to handle the actual crossing network communication between app classes. A method invocation from class *X* to class *N* is passed first from *X* to *NProxy* (which is represented as *NIntf*), then to the endpoint, and finally to *N*. An example of such method forwarding chain is shown in Figure 9. In a method body of *NProxy*, the "invokeMethod" method of the endpoint is used to forward the method invocation to the may-be-offloaded class *N*. The parameters of "invokeMethod" are: (1) the class *N*'s full qualified class name; (2) the instance ID of class *N*. Each *NProxy* holds an ID of a class instance *N*. The ID is initialized when this proxy is created for the corresponding app class instance. Therefore, the endpoint can use the class name and ID to locate the *N* instance uniquely; (3) the bytecode-level method signature. Java bytecode uses "internal name" [41] to represent a class type as shown in Figure 9. By using such internal names, DPartner can easily locate the method being invoked on *N*; (4) the parameters required for the "methodM" of *N* being actually executed.

With the endpoint, the communication between two classes can be easily optimized. For instance, when class *A* and *B* are located in different VMs, the endpoint has to pass method invocations through the network stack (e.g., TCP/IP). When *A* and *B* are both located in the same VM,

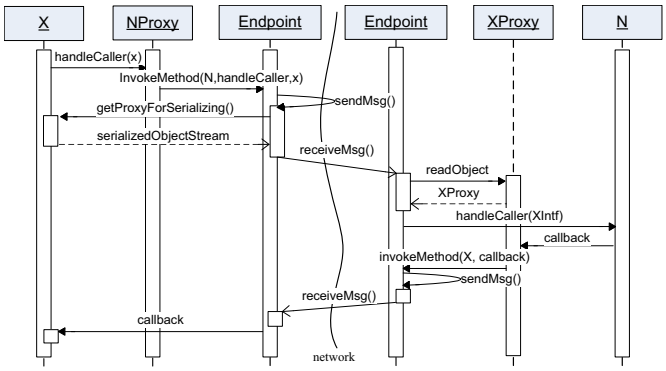


**Figure 10.** Handling callback-style remote invocation

the endpoint will use the in-VM local object reference to forward method invocations from *A* to *B*. In this way, the time-consuming network message transmission is avoided.

When invoking methods through the network stack, the method parameters have to be serialized. We should pay attention to the de/serialization mechanism of the caller object in the callback-style method invocation. For instance, the class instances of *X* run on the phone, and the class instance of *N* run on the server. An *X* instance calls the "handleCaller(XIntf)" method of *NIntf* (which represents an *NProxy* instance when the method is called) by passing itself as the method parameter. The invocation will cause the *X* instance be serialized on the phone side and then be deserialized on the server side. What should be noted is that after deserialization, there will be two *X* instances, one is the original *X* instance on the phone, and the other is the copied *X* instance on the server. If *N* processes the copied *X* on the server, the operations cannot be forwarded back to the *X* on the phone, which will make these two *X* inconsistent. Additionally, if *X* is anchored, it cannot run on the server natively. To solve the above problems, the serialization of *X* should make an *XProxy* instance being serialized instead of the *X* instance itself. Any invocations on the *XProxy* instance will be called back to the *X* instance on the phone.

The serialization of the caller *X* to *XProxy* should not impact the normal serialization operation of *X*. For instance, when *X* is normally serialized, it may be designed to store its fields into the file system. To differentiate these two kinds of serialization operations, DPartner (1) first makes *X* implement a unique interface: *ServerObject* as described in Section 3.3; (2) then creates a unique method "getProxyForSerializing" in *X* through bytecode rewriting. In this method, an *XProxy* will be created and associated with the *X* object through instance ID. This proxy object will be used as the return value of the method; (3) the endpoint uses a specific ObjectStream for object serialization. If an object is an instance of *ServerObject*, the "getProxyForSerializing" method of the object will be invoked. The returned proxy object will finally be serialized and sent to the server. The procedure of such callback-style method invocation is shown in Figure 10.

In addition to forwarding method invocations, the endpoint is also responsible for the context synchronization of class instances being offloaded. As described in Section 2.3, the general offloading procedure is to activate an object on the server/phone, deactivate its counterpart on the phone/server, and synchronize the object state from one side (e.g., phone) to another (e.g., server). The getter/setter methods of the offloaded object are used to collect and inject its states as described in Section 3.3. Dpartner collects class relationships using proxies, and tracks newly created objects using the "reflection" mechanism [42] together with a hashtable-like data structure. Additionally, the specific "SmartObjectInput/SmartOutputStream" are used to check whether a referred object of the offloaded one is serializable. If not, these streams will un/wrap the object using "NotSerializableObjWrapper", which tries to de/serialize the object's fields recursively, and send the serialized data to the other side to keep context synchronized.

Other communication related code, e.g., call-try, deadlock remover [21], and distributed garbage collection [31] are also implemented in the endpoint to form an interception chain for method invocation, which can greatly help improve the quality of the crossing network communication. The details can be found at the project website [8].

## 4. Evaluation

The goals of the evaluation are to (1) validate whether DPartner is applicable to offload real-world apps with reasonable costs; (2) compare the performance of offloaded apps with the original ones; (3) compare the battery power consumption of offloaded apps with the original ones; (4) test whether on-demand offloading can really benefit the phone users.

### 4.1 Experiment Setup

Currently, Wi-Fi is widespread in China for regions like schools, hospitals, malls, etc. For instance, The *China TeleCom corp.* has setup thousands of free Wi-Fi hotspots across the city of Beijing [34]. Therefore, we evaluate DPartner mainly with Wi-Fi connectivity. The tested smartphone is an HTC tattoo [28] with 528MHz CPU, 256MB RAM and Android 2.1. The server is a PC running Ubuntu 8.04 with 2.1GHz dual-core CPU and 1GB RAM. The phone and the server are connected by Wi-Fi with 50ms RTT (Round Trip Time). The server uses NDIS intermediate driver [32] to add a controlled amount of queuing delay to the network path, which helps simulate different RTTs between the phone and the server for evaluating the offloading effects under different network conditions. We measure the battery power consumption of the phone by the PowerTutor Android app [29], which gives the details of the power consumption for each targeted Android app.

We evaluate DPartner on three real-world Android apps shown in Table 2. As described in [7] and [10], the apps worth offloading are usually the computation intensive ones

**Table 2.** The Android apps for evaluation

| Features | Linpack | Chess Game | 3D Car Game |
|---|---|---|---|
| UI | GUI | GUI | GUI |
| Interactive | No | Yes | Yes |
| Computation-Intensive | High | High | High |
| Data-Intensive | Low | Medium | High |
| Multi-Thread | No | Yes | Yes |

such as the image/audio/text processing apps and games, which fall into the following Android Market categories: entertainment and games, media and video, music and audio, books and texts, and photography. More than 50% of 487,601 apps on Android Market in July 2012 [33] belong to these categories, and we selected three typical ones for evaluation. Evaluations on other apps are available on the project website [8].

Table 2 shows the features of the three apps that are important to offloading. The first app is the Linpack benchmark [35] that carries out numerical linear algebra computations. The second is a chess game called Andgoid [36], an interactive app that allows a human player to place chess pieces by using the touch screen of the phone. The human and AI player each runs in a separate thread. It is more data-intensive than the Linpack app because when it is the turn of the AI player, all the chess piece positions on the chess board will be transferred to the AI class to compute the position of the next piece. The third is a 3D car game called XRace [37], a more interactive Android app. The car's direction and speed are controlled by the gravity sensor of the phone. It is also a multi-thread program. Comparing with the other two apps, the car game is the most data-intensive because it reads many data to form the 3D racing scenario, and its collision detection logic has to take in many car position data to check whether the car is running out of the road or colliding with the boundary fence. All the above three apps are computation-intensive, which means that they are much more suitable to be offloaded to take advantage of the powerful processing capability of the server.

### 4.2 Performance of Refactoring

Table 3 shows the refactoring performance of DPartner on the three Android apps. The size of the app after refactoring will be increased because DPartner rewrites each app class and generates proxies and interfaces for them. The endpoint code are also linked into the app. Since the size of smartphone's memory keeps growing and Android supports the "memory to SD card" app-installation feature, the increase of app size is acceptable. The refactoring time increases with the total number of bytecode instructions (just like the lines of code) of the Android app, which is also acceptable in practice because refactoring is performed before runtime.

To some extent, the percentage of movable classes implies the possibility and flexibility of computation offloading. In our experiment, the Linpack app has only 1 movable class, i.e., the Linpack.class, which performs algebra computations. The chess game has 31 movable classes, which are all used for different AI algorithms. The anchored classes of the above two apps all deal with UI interactions. The car game has 39 movable classes while its anchored classes control the gravity sensor and draw images with Android OpenGL ES library [46].

**Table 3.** The refactoring performance of DPartner

| Measurements | Linpack | Chess | Car |
|---|---|---|---|
| Original app size (KB) (.apk file) | 75 | 233 | 12431 |
| The app size after refactoring (KB) | 107 | 306 | 13216 |
| Size increased (KB) | 32 | 73 | 785 |
| # of classes in the original app (#: number) | 36 | 43 | 72 |
| # of classes after refactoring (excluding the classes that make up the app endpoint) | 101 | 152 | 246 |
| # of generated proxies | 36 | 43 | 72 |
| # of generated interfaces and other classes | 29 | 66 | 102 |
| # of classes that make up the app endpoint (using the TCP/IP communication service) | 53 | 53 | 53 |
| Total # of increased classes | 118 | 162 | 227 |
| Total # of methods in the original app | 75 | 177 | 541 |
| Total # of methods after refactoring | 216 | 539 | 1642 |
| Total # of bytecode instructions in the class methods of the original app | 8219 | 16911 | 86524 |
| Total # of bytecode instructions in the class methods of the refactored app | 12982 | 28905 | 110538 |
| # and % of movable classes | 1 (2.8%) | 31 (72.1%) | 39 (42.2%) |
| Refactoring time (s) | 20.2 | 43.7 | 219.8 |

In the movable classes, the most computation intensive one is *com.greenecomputing.linpack.**Linpack*** in Linpack, *insa.android.andgoid.strategy.**MonteCarloAI*** in the chess game, and *com.sa.xrace.collision.**Line2f*** in the car game. These three classes will be monitored by the endpoint at runtime as described in the last paragraph of Section 3.4. Table 4 shows the movable clusters containing the monitored classes. For each app, the movable clusters are arranged in the hierarchical structure just like Figure 8. We manually checked the clustering results and found that all these classes were indeed closely related, which shows the correctness of clustering. Take the 3D car game as a example, the classes such as *Line2f*, *AABBbox*, and *MathUtil* are all used together for collision detection.

### 4.3 Comparison of App Performance

We compared the performance of the above three apps by running them in eight different scenarios. In the first scenario, the original apps run entirely on the phone. In the next scenario, the refactored apps run entirely on the phone. In the following four scenarios, the clustered classes in Table 4 are offloaded to *always* execute on the server. The phone and the server are connected using Wi-Fi with different RTT values (50ms, 100ms, 150ms, and 200ms). The exact clusters to be offloaded are at Level-i, Level-j, and Level-k for the three

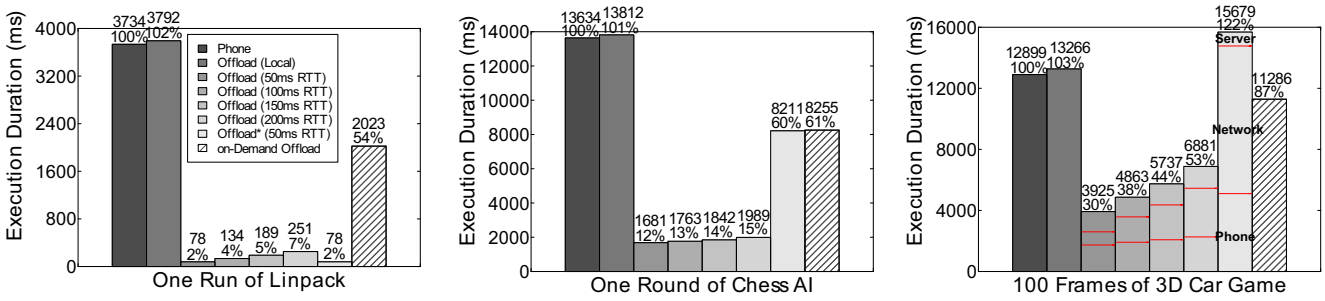


**Figure 11.** The performance comparison of the Android apps running in different scenarios. Offload* is a special offloading test: we force apps to offload only the class that is the most computation intensive, and the RTT value of the test is 50ms

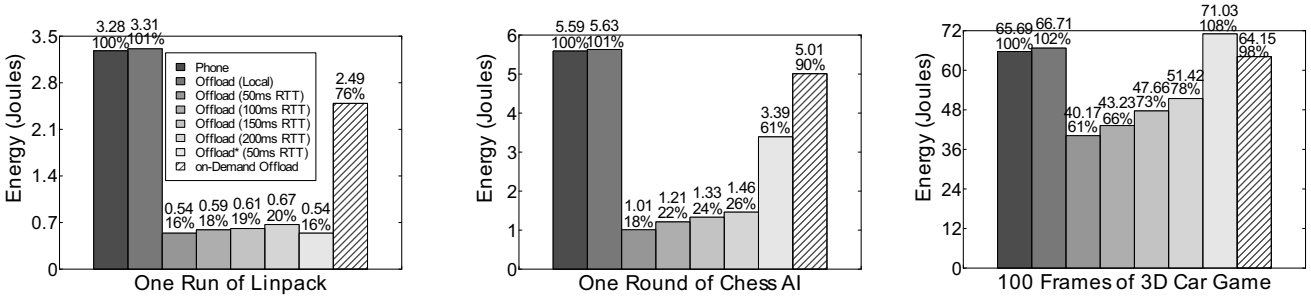


**Figure 12.** The power consumption comparison of the Android apps running in different scenarios

**Table 4.** The movable cluster containing the monitored class

| App | Cluster | Level | The Classes in the Movable Cluster |
|---|---|---|---|
| Linpack | Linpack | Level-i (i≥1) | {**Linpack**} |
| Chess | AI | Level-j (j≥1) | {**MonteCarloAI**, UCTNode} |
| | | Level-(j+1) | {**MonteCarloAI**} |
| Car | Collision | Level-k (k≥1) | {**Line2f**, AABBbox, Rectangle, MathUtil, Matrix4f, Point2f, Point3f, Plane3D, CollisionHandler} |
| | | Level-(k+1) | {**Line2f**, Rectangle, MathUtil, Matrix4f, Point2f, Point3f, Plane3D} |
| | | Level-(k+2) | {**Line2f**, Rectangle, MathUtil, Point2f} |
| | | Level-(k+3) | {**Line2f**} |

apps, respectively. In the seventh scenario, we force apps to offload only the monitored class, i.e., *Linpack*, *MonteCarloAI*, and *Line2f* executed on the server during the test with 50ms RTT, so that we can test whether clustering can really help avoid the high overhead of the crossing network communication. The final scenario is for adapting the offloading and will be discussed in Section 4.5.

The performance comparison results are shown in Figure 11. We can see that, running the refactored app entirely on the phone will increase the execution duration slightly compared with the original app. For instance, in the Linpack test, one run of the original Linpack costs 3.734s, while the refactored Linpack costs 3.792s, i.e., with an overhead of 0.058s. For another instance, the time cost for drawing 100 frames of the refactored 3D car game running entirely on the phone is 0.367s slower than the original app. The slight increase of execution time is due to that method invocations between local classes will be forwarded by the proxies and the endpoint.

However, offloading can really help improve the app performance. For instance, when offloading the Linpack cluster to the server, the execution time of the same test with 50ms RTT is just 0.078s or reduced by 98%. The time for the AI to calculate the chess piece position in the original chess game is 13.63s on average, while the time for the refactored chess game with the AI cluster being offloaded is just 1.681s or reduced by 88%. The execution time for drawing 100 frames in the car game is reduced by 70%. The reason for the great performance improvement is that the computation intensive code is executed on a more powerful processor of the server other than the phone's own processor.

The quality of the network impacts the offloading effect significantly. If the RTT value becomes larger, the performance of the offloaded app will get decreased due to the fact that more time will be spent on network communication. For instance, when RTT is 50ms, drawing 100 graph-frames in the 3D car game will cost 3.925s. However, when RTT is 200ms, this value is just 6.881s or is increased by 75%.

Offloading only the most computation intensive class but not the class cluster it belonged to will often put a negative impact on the app's performance, and even make offloading unworthy. For instance, in the "Offload*" test of the chess game, the time cost for the AI player to calculate the chess piece position is 8.211s; while offloading the AI cluster (i.e., the cluster containing the *MonteCarloAI* and the *UCTNode*

classes) can cost only 1.681s to finish such a computation. For another instance, in the "Offload*" test of the 3D car game, drawing 100 frames will cost 15.679s, which is even longer than that of the original app running on the phone. The total time can be divided into three parts: (1) *phone*, the time spent on the phone; (2) *network*, the time spent on the network; (3) *server*, the time spent on the server. The time spent on the network increases sharply in "Offload*" (accounting for 63% of the total execution time of this test, as shown in Figure 11). Thus we can see that offloading in the cluster unit can greatly improve the performance of an offloaded app, because it reduces the unnecessary network communication between the parts on the phone and the parts on the server.

### 4.4 Comparison of App Power Consumption

The power consumption results are shown in Figure 12. When running the refactored app entirely on the phone, its power consumption will increase slightly compared with the original app, because the proxies and the endpoint in the refactored app cost energy to run. We can also see that, as offloading makes some computation intensive code be executed on the server, the energy consumption of an offloaded Android app is often reduced. For instance, in the Linpack test, running the benchmark one time will cost 3.28 Joules, while the result of the offloaded Linpack with 50ms RTT is just 0.54 Joules or reduced by 83%. What should be noted is that, the power of the Wi-Fi on the HTC smartphone we used is about 0.78 watt [28], which may make the offloaded app's power consumption be greater than that of the original app. However, we get very small energy consumption (e.g., 0.54J) in the offloaded tests because: (1) when the Wi-Fi device is not working, it will enter a sleep phase for saving energy. The power consumption in such a phase is about 30mJ on the tested smartphone [28]; (2) In the Linpack app, one execution of the benchmark will run the *Linpack* class instance 3 times to get the average test results. The number of runs (i.e., 3) will be sent as a parameter to the *Linpack* class. Therefore, in the offloaded app, the Wi-Fi channel will deliver the method request for linpack-calculation just once, and the server will execute the *Linpack* class instance three times before returning the average results. Thus the power consumption of the Wi-Fi is further reduced; (3) the time for sending one method request is very short, e.g., usually less than 10ms. Therefore, the total power consumed by Wi-Fi will not be much when the number of the crossing network communication is relatively small. However, as shown by the "Offload*" test in Figure 12, if the network communication becomes too frequent, the power consumption of Wi-Fi will increase. Thus the whole power consumption of the offloaded app will also increase. For instance, when offloading only the *Line2f* class of the 3D car game, the app will consume 30.86 more Joules than offloading the whole *Collision* class cluster. A lot of energy is wasted on the crossing network communication between the closely-related but not-together-offloaded classes.

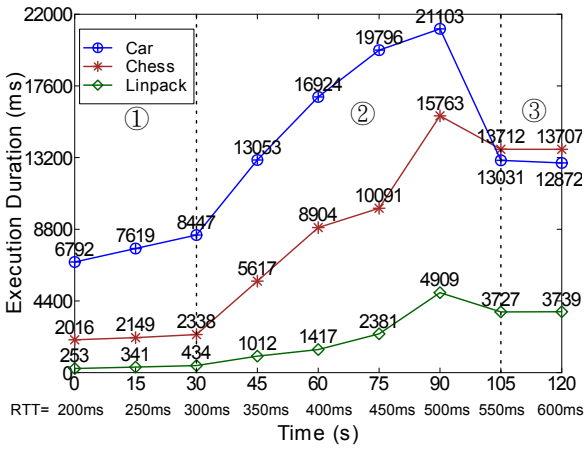


**Figure 13.** The app performance during on-demand offloading

**Table 5.** The on-demand offloaded computations in detail

| Time (s) | RTT (ms) | Offloaded Computations | | | Description |
|---|---|---|---|---|---|
| | | Linpack | Chess | Car | |
| 0 | 200 | Level-i | Level-j | Level-k | offloaded execution, performance decreases with the increase of RTT |
| 15 | 250 | Level-i | Level-j | Level-k | |
| 30 | 300 | Level-i | Level-j | Level-k | |
| 45 | 350 | Level-i | Level-j | Level-k | network packet loss, invocation retry, from offloading only the suitable computations to offloading none |
| 60 | 400 | Level-i | Level-(j+1) | Level-(k+1) | |
| 75 | 450 | Level-i | Level-(j+1) | Level-(k+2) | |
| 90 | 500 | Level-i | Level-(j+1) | Level-(k+3) | |
| 105 | 550 | None | None | None | |
| 120 | 600 | None | None | None | local execution |

### 4.5 The Effect of On-Demand Offloading

The last scenario in Figures 11 and 12 shows the performance and power consumption of the three apps when offloading is adapted during the RTT value changing from 200ms to 600ms gradually in 120 seconds. The corresponding details are presented in Figure 13 and Table 5. We can see that the performance plot of the apps can be divided into 3 regions. Region-1 is 0∼30s when RTT increases from 200 to 300ms gradually. During this period, the offloaded computations are the Level-i cluster of Linpack, the Level-j cluster of chess, and the Level-k cluster of car game (see Table 4). The app performance decreases with the increase of RTT. Region-2 is 30∼105s when RTT increases from 300 to 550ms. The app performances during this period fall sharply. Network packet loss has happened in this period, which causes the endpoint to retry method invocation several times. When found that the offloaded computations fail to satisfy the offloading requirements described in Section 3.5, the endpoint will draw these computations back to execute on the phone. For example, the endpoint of the car game will first draw back the computations of the classes in cluster

(Level-k) - cluster (Level-(k+1)), that is, the class instances belonging to cluster (Level-k) but not to cluster (Level-(k+1) will be drawn back to execute on the phone (see Table 4 and 5). At this point, the offloaded computations are just the class instances of the cluster Level-(k+1). As RTT increases continuously, the endpoint will change the contents of the offloaded computations to maintain the performance as described in Section 3.4. For example, the offloaded computations of the car game changes from Level-(k+1) to Level-(k+2), then to Level-(k+3), and finally to none. In the 105th seconds, all the computations of the tested apps have come back to execute on the phone till the end of the test as shown in Region-3 of the plot.

### 4.6 Experiments on 3G Network

As 3G has become a widely accepted way to connect a smartphone and the servers in Cloud, we continue to test the refactored apps with 3G connectivity.

We use a server in our Cloud platform [47], which can be publicly accessed using 3G. The server is a Ubuntu 8.04 Xen VM [48] with 2.0GHz four-core CPU and 1GB RAM running on IBM blade HS22. We use the Chess app in the experiment with the experiment setup remained the same as previously, except that the offloaded cluster is accessed using 3G. The remained parts of the app on the phone use HTTP instead of TCP connections to interact with the offloaded parts of the app running on the server. The RTT value of the 3G test is about 360 ms, the upload speed is 23.4kb/s, and the download speed is 28.9kb/s.

Figures 14 and 15 present the performance and power consumption results, respectively. We have three observations from the results. First, offloading can really help improve the app performance a lot even if we connect the phone with the server using a relatively slower network connection of 3G instead of Wi-Fi. The average execution time of "one round of Chess AI" is 2.426s or reduced by 82.2% in 3G test compared with that of the original app running entirely on the phone. Second, The performance of the server greatly affects the offloading effects. The Cloud server we used in the experiment is much more powerful than the PC we used in the prior tests. Therefore, the time delay brought by the slow network connection of 3G can be partly offset by the high computing capability of the Cloud server, and the finally obtained offloading effects are still very good. Third, 3G costs more energy than Wi-Fi. The power consumption of "one round of Chess AI" with Wi-Fi is 1.01 Joules on average. The consumed energy increases to 2.56 Joules in the 3G test. The above results indicate that offloading can save energy and improve performance for smartphone apps. However, this usually depends on the server's performance and the amounts of data exchanged through the network. Therefore, computation offloading is more suitable to offload computation-intensive apps with only a small amount of data exchanged between the server and the client [7], especially on 3G network.

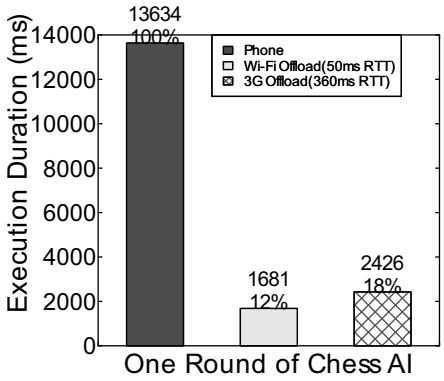


**Figure 14.** The chess's performance in 3G test

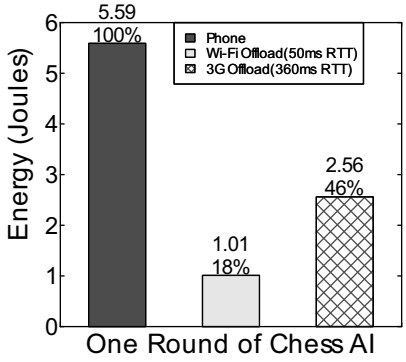


**Figure 15.** The chess's power consumption in 3G test

## 5. Related Work

The idea of using a strong server to enhance the processing capabilities of a weak device (e.g., mobile phone) is not new [7]. Many early research projects have tried to automatically partition a standalone desktop app to have some parts of it executed remotely on the server. The research on mobile computing then leverage such an idea to realize computation offloading on a smartphone.

Coign [12] is an early work on computation offloading. It takes the binary code of a Windows .COM application, then by using code interception and rewriting, it turns the app into a DCOM application with some COM components running on a common PC (i.e., a weak device) and the rest running on a powerful server (i.e., a strong surrogate). In this way, the performance of the original application can be improved. J-Orchestra [13] and JavaParty [14] are the early work on offloading standalone desktop Java applications. J-Orchestra works on the Java bytecode level, while JavaParty works on the source code level. They both require developers to manually tell the offloading tool about which class can be offloaded. For instance, J-Orchestra provides a GUI to ask developers to select the can-be-offloaded classes. JavaParty asks developers to annotate the source code with the "Remote" keyword. Then the compiler provided by these tools will compile these selected classes to generate the RMI

stubs/skeletons for them [45], so that the app is turned to be a client/server one by using RMI as the communication channel. The work of J-Orchestra and JavaParty cannot be directly used for offloading Android apps. One reason is that Android does not support RMI.

The research on mobile computing follows the above early work to offload computations running on the phone to run on the server. AIDE [15][16] can partition a mobile Java application at runtime through JVM instrumentation. It leverages a fuzzy control model to decide which class should be transferred to the server. The code on the phone and the code on the server can cooperate to work with the support of the modified JVM. Similarly, CloneCloud [17] augments the performance of Smartphone apps through *clone cloud execution*. It modifies the Android Dalvik VM to provide an application *partitioner* and an *execution runtime* to help apps running on the VM offload tasks to execute on a cloned VM hosted by a Cloud server. MAUI [11] requires developers to annotate the can-be-offloaded methods of a .Net mobile application by using the "Remoteable" annotation. Then the MAUI analyzer will decide which method should really be offloaded through runtime profiling. Cuckoo [10] is an offloading framework for Android apps. It requires developers to follow a specific programming model to make some parts of the app be offloaded. JDOP [18] mainly focuses on the algorithm of how to distribute objects among the phone and the server to achieve high app performance, and provides the request resolver and dispatcher facilities to make objects be able to offload. Spectra [19] requires developers to specify movable classes and modify the application. It then performs offloading at the granularity of methods. Puppeteer [20] targets at data adaptation via offloading when facing with limited bandwidth, and it is applicable for only COM/DCOM-like component-based applications.

Our offloading approach and the DPartner tool are different from the above work mainly in the aspects below. First, our approach is transparent to developers. We neither require them to annotate the code of an Android app to decide which class should be offloaded, nor require them to follow a specific programming model to redesign the whole app. Second, offloading at the granularity of class/object makes our approach be more suitable for object-oriented programs. Third, our approach can enable the on-demand offloading of an given Android app, while few of the above existing work can support such a feature. What should be noted is that, although the research projects such as AIDE, CloneCloud, and MAUI have tried to make offloading be on-demand. They all have some obvious drawbacks that can make them be impractical. For instance, AIDE and CloneCloud require using a modified JVM, and MAUI requires developers to annotate source code methods. DPartner does not impose such requirements. It can refactor any Android app into the one supporting on-demand offloading, no matter the app is newly designed or is already installed on a phone.

## 6. Conclusion and Future Work

In this paper, we presented DPartner for automatically refactoring an Android app into one implementing the on-demand computation offloading design pattern, which can transfer some computation-intensive tasks from a smartphone to a server so that the task execution time and battery power consumption of the app can be reduced significantly. DPartner will be improved and further validated as follows. First, more experiments with real-world Android apps will be done, especially on 3G network. Second, to support refactoring and offloading obfuscated apps. DPartner leverages Dexmaker [43] and ASM [44] for bytecode refactoring. These libraries do not fully support obfuscated bytecode currently. Third, better measuring tools will be used. For example, we will use an external current meter to measure the power consumption of a phone much more accurately, and use routers to measure the amount of network packets transmitted. Fourth, more rules and algorithms will be explored and evaluated for deciding which class should be offloaded. We will put further results on the website of DPartner: https://code.google.com/p/dpartner/.

## Acknowledgments

We thank Professor Lu Zhang of Peking University, Professor Tao Xie of North Carolina State University, Professor Zhendong Su of the University of California - Davis, and the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 61121063, 60933003, 61003010; the European Commission Seventh Framework Programme under grant no. 231167; the IBM-University Joint Study and the NCET.

## References

[1] Android, http://www.android.com/

[2] Smartphone Market Share, http://www.idc.com/getdoc.jsp?containerId=prUS23503312

[3] Google Play, https://play.google.com/store

[4] The applications number of the Google Play, http://en.wikipedia.org/wiki/Google_Play

[5] Apps drain battery power, www.droidforums.net/forum/droid-razr-support/216454-battery-drain.html

[6] Android Application Requirements, www.netmite.com/android/mydroid/development/pdk/docs/system_requirements

[7] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, Bharat Bhargava. "A Survey of Computation Offloading for Mobile Systems". In *the Journal of Mobile Networks and Applications*, Springer, April, 2012.

[8] Dpartner: http://code.google.com/p/dpartner/

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. "Refactoring: Improving the Design of Existing Code". *Publisher:Addison-Wesley*, First edition, 1999.

[10] Roelof Kemp, Nicholas Palmer. “Cuckoo: a Computation Offoading Framework for Smartphones”. In *Proceedings of the International Conference on Mobile Computing, Applications, and Services* (MobiCase), pp. 1-20, 2010.

[11] Eduardo Cuervoy, Aruna Balasubramanianz, Stefan Saroiux, Ranveer Chandrax, Paramvir Bahlx. “MAUI: Making Smartphones Last Longer with Code Offload”. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (MobiSys), pp. 49 - 62, 2010.

[12] Galen C. Hunt and Michael L. Scott. “The Coign Automatic Distributed Partitioning System”. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pp. 187-200, 1999.

[13] Eli Tilevich and Yannis Smaragdakis. “J-Orchestra: Enhancing Java Programs with Distribution Capabilities”. In *ACM Transactions on Software Engineering and Methodology* (TOSEM), Vol. 19, No. 1, Article 1, pp. 1-41, 2009.

[14] Michael Philippsen and Matthias Zenger. “JavaParty: Transparent Remote Objects in Java”. In *Concurrency: Practice and Experience* 9(11):1225-1242, John Wiley & Sons, Ltd., 1997.

[15] Alan Messer, Ira Greenberg, Philippe Bernadat, DeJan Milojieie, Deqing Chen, T.J. Giuli, Xiaohui Gu. “Towards a distributed platform for resource-constrained devices”. In *Proceedings of the International Conference on Distributed Computing Systems* (ICDCS), pp. 43-51, 2002.

[16] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. “Adaptive Offloading for Pervasive Computing”. In *IEEE Pervasive Computing*, pp. 66-73, 2004.

[17] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, et al. “CloneCloud: Elastic Execution between Mobile Device and Cloud”. In *Proceedings of the European Conference on Computer Systems* (EuroSys), pp. 301-314, 2011.

[18] Lei Wang and Michael Franz. “Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives”. In *Proceedings of the International Conference on Parallel and Distributed Systems* (ICPADS), pp. 369-376, 2008.

[19] J. Flinn, S. Park, and M. Satyanarayanan. “Balancing Performance, Energy, and Quality in Pervasive Computing”. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (ICDCS), pp. 1-10, 2002.

[20] E. Lara, D. S. Wallach, and W. Zwaenepoel. “Puppeteer: Component-based Adaptation for Mobile Computing”. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (USITS), pp. 159-170, 2001.

[21] Eli Tilevich and Yannis Smaragdakis. “Portable and Efficient Distributed Threads for Java”. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference* (Middleware), pp. 478-492, 2004.

[22] Girvan, M. and NewMan, M.E.. “Community structure in social and biological networks”. In *Proceedings of the National Academy of Sciences of the United States of America* (PNAS), Vol. 99, pp. 7821-7826, 2002.

[23] J. Han and M. Kamber. “Data Mining: Concepts and Techniques”. *Morgan Kaufmann Publishers*, 2001.

[24] Lu Zhang, Jing Luo, He Li, Jiasu Sun, and Hong Mei. “A Biting-Down Approach to Hierarchical Decomposition of Object-Oriented Systems Based on Structure Analysis”. In *the Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 22, No. 8, pp. 567-596, 2010.

[25] Jonathan I. Maletic and Andrian Marcus. “Supporting Program Comprehension Using Semantic and Structural Information”. In *Proceedings of the International Conference on Software Engineering* (ICSE), pp. 103-112, 2001.

[26] Call Graph, http://en.wikipedia.org/wiki/Call_graph

[27] Walter Binder, et al. “Using Bytecode Instruction Counting as Portable CPU Consumption Metric”. In *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 57-77, 2006.

[28] HTC Tattoo, www.htc.com/europe/product/tattoo/overview.html

[29] PowerTutor, http://powertutor.org/

[30] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, et al. “Accurate online power estimation and automatic battery behavior based power model generation for smartphones”. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS), pp. 105-114, 2010.

[31] Saleh E., Abdullahi, et al. “Garbage Collecting the Internet: A Survey of Distributed Garbage Collection”. In *ACM Computing Surveys*, Vol. 30, No. 3, pp. 330-373, 1998.

[32] Network Driver Interface Specification (NDIS), http://wikipedia.org/wiki/Network_Driver_Interface_Spec

[33] The Android application categories, http://www.appbrain.com/stats/android-market-app-categories

[34] Wi-Fi Hotspots in Beijing, http://www.theregister.co.uk/2011/11/03/china_free_wifi/

[35] Linpack, https://market.android.com/details?id=com.greenecomputing.linpack

[36] Chess Game, http://code.google.com/p/andgoid/

[37] Car Game, http://code.google.com/p/xrace-sa/

[38] Android Service, http://developer.android.com/reference/android/app/Service.html

[39] Android Interface Definition Language (AIDL), http://developer.android.com/guide/developing/tools/aidl.html

[40] Android Api Demo, http://developer.android.com/resources/samples/ApiDemos/

[41] The $Java^{TM}$ Virtual Machine Specification, second edition, http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[42] Java Reflection, http://java.sun.com/developer/technicalArticles/ALT/Reflection/

[43] Dexmaker: Programmatic code generation for Android, http://code.google.com/p/dexmaker/

[44] ASM: A Java bytecode engineering library, http://download.forge.objectweb.org/asm/asm4-guide.pdf

[45] Remote Method Invocation (RMI), http://docs.oracle.com/javase/tutorial/rmi/overview.html

[46] Android OpenGL ES, http://developer.android.com/guide/topics/graphics/opengl.html

[47] Internetware Testbed, http://icloud.internetware.org/

[48] Xen, http://www.xen.org/