Code Offload with Least Context Migration in the Mobile Cloud

Yong Li, Wei Gao
Department of Electrical Engineering and Computer Science
University of Tennessee at Knoxville
{yli118,weigao}@utk.edu

***Abstract*—Mobile Cloud Computing (MCC) is of particular importance to address the contradiction between the increasing complexity of user applications and the limited lifespan of mobile device's battery, by offloading the computational workloads from local devices to the remote cloud. Current offloading schemes either require the programmer's annotations, which restricts its wide application; or transmits too much unnecessary data, resulting bandwidth and energy waste. In this paper, we propose a novel method-level offloading methodology to offload local computational workload with as least data transmission as possible. Our basic idea is to identify the contexts which are necessary to the method execution by parsing application binaries in advance and applying this parsing result to selectively migrate heap data while allowing successful method execution remotely. Our implementation of this design is built upon Dalvik Virtual Machine. Our experiments and evaluation against applications downloaded from Google Play show that our approach can save data transmission significantly comparing to existing schemes.**

I. Introduction

Smartphones nowadays are designated to execute computationally expensive applications such as gaming, speech recognition, and video playback. These applications increase the requirements on smartphones' capabilities in computation, communication, and storage, and seriously reduce the smartphones' battery lifetime. Mobile Cloud Computing (MCC) [1] could be a viable solution to bridge the gap between limited capabilities of mobile devices and the increasing users' demand of mobile multimedia applications, by offloading the computational workloads from local devices to the cloud.

Due to the expensive wireless communication between smartphones and the remote cloud through cellular or WiFi networks, a mobile application needs to be adaptively partitioned according to the computational complexity and size of operational datasets of different application methods, so as to ensure that the amount of energy saved by remote execution overwhelms the expense of wirelessly transmitting the relevant application datasets to the remote cloud [2]. Intensive research has been conducted on how to appropriately decide such application partitioning [3], [4], [5], [6], and support remote application execution through techniques of code migration [4], [5] and Virtual Machine (VM) synthesis [7], [8], [9]. However, these traditional schemes either restrict the scope of workload offloading to a specific set of system frameworks and mobile applications [4], [5], or migrate a large amount of application contexts to the remote cloud regardless of the specific execution patterns of the application partition to be offloaded [7], [8]. These limitations seriously impair the efficiency of workload offloading in practical mobile cloud scenarios, which is challenging to be further improved due to the following reasons.

This work was supported in part by the DOD Grant W911NF-14-1-0262.

First, the computational workloads at local mobile devices needs to be offloaded automatically by the mobile Operating System (OS) without programmers' intervention, so that the large population of existing mobile application executables can be efficiently partitioned and offloaded for remote execution without any modification or additional efforts of software redevelopment. To address this challenge, the offloading engine must be integrated with the OS kernel level and directly interacts with the intact application binaries. Existing offloading schemes, in contrast, rely on the application developers' offline efforts to declare the sets of application methods to be offloaded [4], [5], and lack of the capability of run-time application partitioning and profiling [7].

Second, only the memory contexts that are relevant to the current application methods being offloaded should be migrated to the remote cloud. Some existing code migration systems [7], [8] suggest to migrate only the thread reachable contexts to reduce the amount of wireless data transmission from unconscious migration of the full application process. However, many irrelevant contexts that reside in the application stack or memory heap of the executing thread may still be migrated without discretion.

In this paper, we present a novel design of workload offloading system which addresses the aforementioned challenges and performs automated method-level workload offloading with least context migration. Our basic idea of achieving the least context migration while ensuring the offloading appropriateness is to identify the memory contexts that may be accessed by a specific application method prior to its execution, through offline parsing of the application executables. The parsing results will be stored as metadata along with the application executables at local mobile devices, and will be utilized by the run-time application execution to screen the thread stack and heap contexts to migrate only the relevant memory contexts to the remote cloud. We have implemented the proposed system design over practical Android systems, and the experimental results over realistic smartphone applications show that our system can migrate 70% less memory contexts compared to existing schemes, while maintaining the same offloading effectiveness. To the best of our knowledge, we are the first to exploit the inner characteristics of application binaries for workload offloading in mobile clouds.

Our detailed contributions are as follows:

- We develop a systematic approach to identify the memory contexts which may be accessed by each method during its execution through offline parsing to application binaries. The parsing results can then be used as metadata for remote method execution.
- We propose a novel way to reduce the wireless data traffic of workload offloading by applying the metadata on the dynamic heap contexts at run-time. The subsequent context migration hence minimizes the amount of irrelevant memory contexts being involved.

The rest of this paper is organized as follows. Section II reviews the existing work. Section III-A introduces the Android system background related to our offloading system. Section III-B describes the motivation for our work, and Section III-C presents our high-level system design. Sections IV and V present the technical details of our proposed techniques of offline parsing and run-time migration. Section VI evaluates the performance of our system. Section VII discusses and Section VIII concludes the paper.

II. Related Work

Workload offloading in MCC focuses on addressing the problems of *what* to offload and *how* to offload. A prerequisite to efficient workload offloading is to decide appropriate application partitions. Such decisions are based on the profiling data about application execution and system context, such as the CPU usage, energy consumption, and network latency. Some schemes such as MAUI [4] and ThinkAir [5], which provide a system framework to handle the internal logic of workload migration, rely on developers' efforts to annotate which methods should be offloaded. Other schemes [10], [8], [11] use online profiling techniques to monitor application executions. Based on these profiling data, empirical heuristics with specific assumptions are used to partition user applications. For example, Odessa [12] assumes linear speedup over consecutive frames in a face recognition application. ThinkAir [5] defines multiple static offloading policies, each of which focuses on a sole aspect of system performance. Nevertheless, the major focus of this paper is to develop systematic techniques improving the energy efficiency of workload migration, and is hence orthogonal to the decisions of application partitioning. In our system implementation, we use an online profiler to monitor the methods' execution times, based on which the decisions of workload offloading are made.

Various systematic solutions, on the other hand, are developed to offload the designated application partitions from local devices to the remote cloud or cloudlets [1], [13]. MAUI [4] wraps the memory contexts of the offloading method into a wrapper, and then sends these contexts through XML-based serialization. Our proposed work, in contrast, migrates the memory contexts as raw data and hence avoids the cost of transmitting the XML tag information. ThinkAir [5] focuses on the scalability of VM in the cloud, but does not focus on the efficiency of VM migration between the local mobile devices and the remote cloud.

CloneCloud [7] and COMET [8], being similar to our proposed system, offload the computational workloads through VM synthesis [14], [15]. CloneCloud [7] is only able to offload one thread of an application process, and hence has limited applicability for current multi-threaded mobile applications. In contrast, our proposed system supports multi-threaded application execution by adopting the Distributed Shared Memory (DSM) [16] technique. Similar technique is also used in COMET [8], which aims to mirror the application VMs from the local devices to the remote cloud by migrating and synthesizing all the reachable memory contexts within the executing threads, but significantly increases the wireless data traffic of workload offloading. In contrast, we propose to only migrate to the remote cloud the memory contexts that are relevant to the corresponding remote method execution. Instead of running a complete duplicate of the local VM, the cloud is only regarded as an execution container for the current method execution.

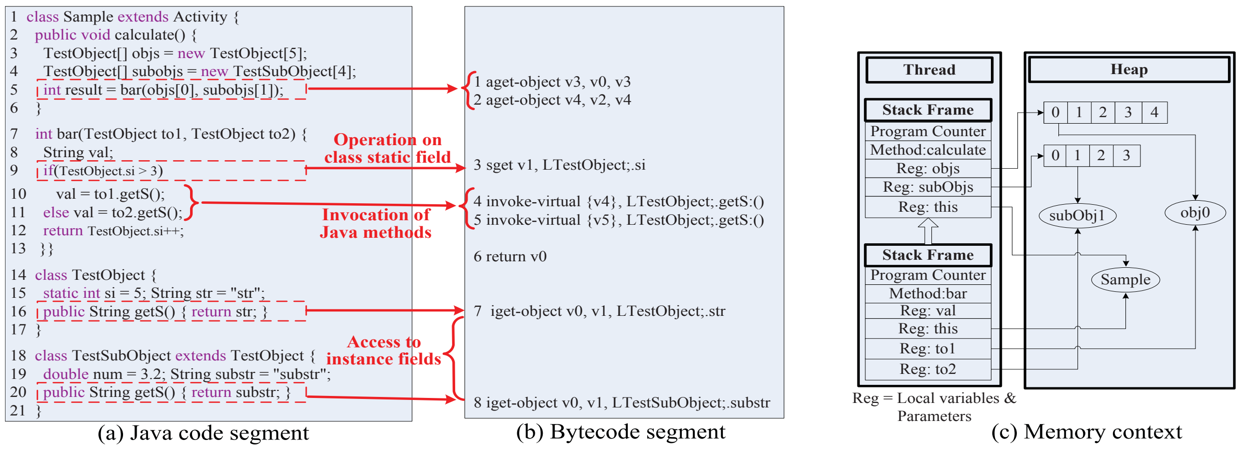
III. Overview

A. Background of Android System

In this section, we briefly introduce the necessary background of the Android system, which is the most popular OS nowadays on various mobile platforms [17], [18] and our targeting system platform throughout this paper. An Android-based mobile application, running as a Dalvik VM, is written in Java. The java source files of an user application are compiled by the Java compiler into Java bytecodes as class files, which are then compressed and translated into register-based Android bytecodes by *dexgen*.

We demonstrate such model of Android system execution using an example of code segment shown in Figure 1. This example will also be used throughout the rest of this paper to illustrate our ideas and system designs. As shown in Figure 1(b), there are three major types of bytecodes that may be generated in an Android application. First, Java method invocation will be converted into *invoke-kind*, such as *invoke-interface* - calling to an interface method, and *invoke-virtual* - invoking a method that can be overridden by subclasses (e.g., the *to1.getS()* method in Line 10 of Figure 1(a)). Second, operations on class static fields (e.g., *TestObject.si* in Line 9 of Figure 1(a)) will be translated into the bytecodes *sget* or *sput*. Third, access to an instance field will be transformed as *iget* or *iput* (e.g., Lines 16 and 20 in Figure 1(a)).

When an application starts, its executable that contains the Android bytecodes, will be loaded into the Dalvik VM which creates a number of application threads for method executions. During method executions, a method may invoke another method. To preserve such method invocation chain, an invocation stack is maintained in each thread, and a stack frame will be associated to each invoking method with a pointer to its invoker. All the information relevant to the method execution, including current Program Counter (PC), method reference and registers, will be stored in the stack frame. For example in Figure 1, when the method *bar()* is being executed, the thread stack and memory heap space is shown in Figure 1(c). The stack frame of *bar()* will point to



```
class Sample extends Activity {
 public void calculate() {
  TestObject[] objs = new TestObject[5];
  TestObject[] subobjs = new TestSubObject[4];
  int result = bar(objs[0], subobjs[1]);
 }
 int bar(TestObject to1, TestObject to2) {
  String val;
  if(TestObject.si > 3)
    val = to1.getS();
  else val = to2.getS();
  return TestObject.si++;
 }}
class TestObject {
 static int si = 5; String str = "str";
 public String getS() { return str; }
}
class TestSubObject extends TestObject {
 double num = 3.2; String substr = "substr";
 public String getS() { return substr; }
}
```

```
aget-object v3, v0, v3
aget-object v4, v2, v4
sget v1, LTestObject;.si
invoke-virtual {v4}, LTestObject;.getS:()
invoke-virtual {v5}, LTestObject;.getS:()
return v0
iget-object v0, v1, LTestObject;.str
iget-object v0, v1, LTestSubObject;.substr
```

(a) Java code segment (b) Bytecode segment (c) Memory context

Fig. 1. An example of the execution model of Android applications

its caller *calculate()*. The arguments and local variables of a method are located in the stack frame as a list of registers. If a variable is a primitive type, its actual value is stored in the frame register. If the variable is a reference type, the value of the frame register will be its address in the memory heap.

B. Motivation

According to the above model of Android application execution, not all the stack information or heap contexts are necessary for a specific application method to execute. Even for the input arguments given to a method, the method may only access a portion of their fields. This observation motivates us to exploit the application binary to find out which portion of memory contexts are required to assure successful remote method execution, so as to only migrate this portion of memory contexts for workload offloading.

We further illustrate such motivation of our proposed work using the example in Figure 1, when the method *bar()* is going to be offloaded for remote execution. In traditional offloading schemes such as COMET [8], in order to offload the method *bar()*, it will not only transmit the stack frame and heap objects of *bar()* to the remote cloud, but will also transmit those of its caller, which is the stack frame of *calculate()*, the arrays *objs[]* and *subobjs[]*, although only one element in each array will be actually accessed by *bar()*. The goal of our work, therefore, is to appropriately migrate only the first element of *objs* and second element of *subObjs*. Furthermore, by parsing the application binary, our work can successfully identify that the field *num* in the argument *to2* has no way to be accessed throughout the execution of *bar()*. Thus, during offloading, we will not migrate the *num* field of *to2*, either.

C. Big Picture

In this section, we will describe our system architecture, as shown in Figure 2, to give a high level overview of our proposed system design. There are two major components in our system. One component is for *Offline Parsing* and the other is for *Run-time Migration*.

The purpose of the *Offline Parsing* component is to identify the relevant heap objects that a method may operate. During the method execution, there are two possible sources of objects it can access. One is the input arguments and the other is the

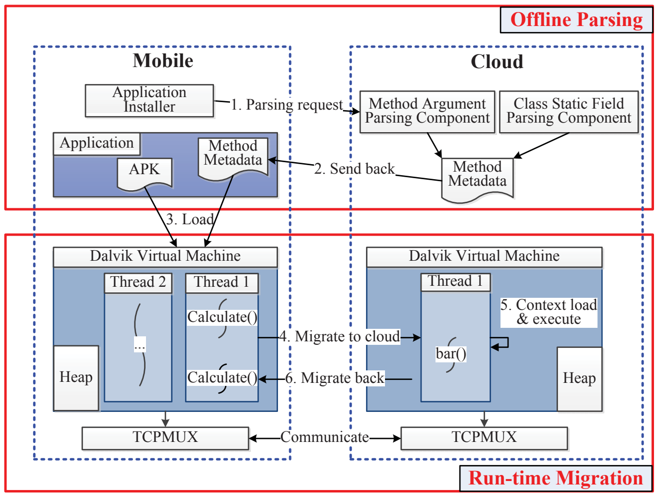


Fig. 2. The system Architecture of the workload offloading system

class static fields. For example, in the method *bar()* of class *Sample* in Figure 1(a), in addition to the input arguments *to1* and *to2*, the method *bar()* also has access to the static field *si* of *TestObject*. As a result, we parse these two types of memory contexts using the *Method Argument Parsing Component* and *Class Static Field Parsing Component*, respectively. Both components will do an offline parsing to identify the migration contexts needed for the method. First, the *Method Argument Parsing Component* is responsible for determining which fields in the input arguments may be accessed during method execution. It goes through all the possible execution paths in the method and tracks the changes of registers to see which field of an object will be accessed in instruction. For example, with an *if/else* condition, this component will parse *if* as a path and *else* as the other path, since the same variable may hold different value after either path is executed, like the local variable *val* of *bar()* in Figure 1(a). Second, the *Class Static Field Parsing Component* is responsible for finding out which class and its static fields may be operated by a method. It does not consider the states of registers in the stack frames, since they are not involved to determine the field of class on which the bytecode instructions are operated. When both of these components are finished, the parsing results will be maintained as metadata, which are sent to the local mobile device and encapsulated along with the corresponding application executable.

When an application is launched at a mobile device, both of its application executable (the .apk file) and method metadata generated by the Offline Parsing component will be loaded by the Dalvik VM, which tracks and profiles all the method invocations during the application execution. When a method is going to be offloaded, its invocation will be intercepted by the *Run-time Migration* component, which then utilizes the corresponding metadata to search for the dynamic heap contexts and determine the necessary contexts for the remote method execution in the cloud. These data objects are migrated to cloud and used to build the run-time environment for remote method execution, which requires loading the heap objects into the memory space, reconstructing the stack frame, and creating an executing thread. When the method execution finishes, the method stack frame and heap objects modified during method execution will then be migrated back to the local mobile device.

IV. Offline Parsing

In this section, we describe the technical details of the Offline Parsing component in our proposed system design. The task of this component is to find out the appropriate part of the input arguments and class static fields that may be operated during the invocation of a specific application method, by parsing the application binaries offline. This task, however, is challenging due to the following reasons. First, the polymorphism feature of the object-oriented Java programming language makes it hard to determine the actual types of memory objects and method invocations prior to the run-time application execution. Our work addresses this challenge by parsing all the possible application cases raised by polymorphism. Second, the program semantics in an application method may significantly vary due to the different combinations of bytecode instructions, and hence complicate the parsing process. This challenge is addressed by analyzing the semantics of every bytecode instruction and emulate its effect to the program registers.

We implement our Offline Parsing component as an independent module targeting for x86 architectures, in Android source with CyanogenMod's Jelly Bean version, and build this module on Linux distribution Ubuntu 12.04. The implementation consists approximately 2,500 lines of C codes. The application executables being parsed are directly downloaded from Google Play instead of being transmitted from the local mobile devices.

A. Method Argument Parsing Component

The major challenge of parsing the method arguments is the diversity of possible application execution path at run-time. Such diversity is generally a result of code polymorphism in Java, as well as the control flow statements in the application source code such as the *if/else* or *switch* clauses and the *for* loops. The number of possible execution paths grows exponentially with the number of program branches and the number of child classes. Take the code segment shown in Figure 1(a) as

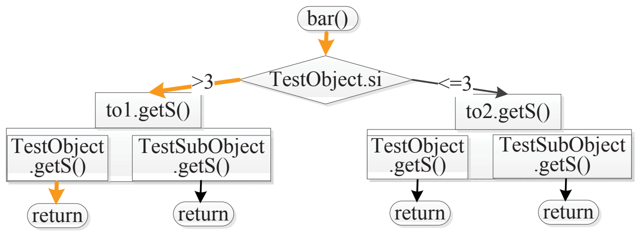


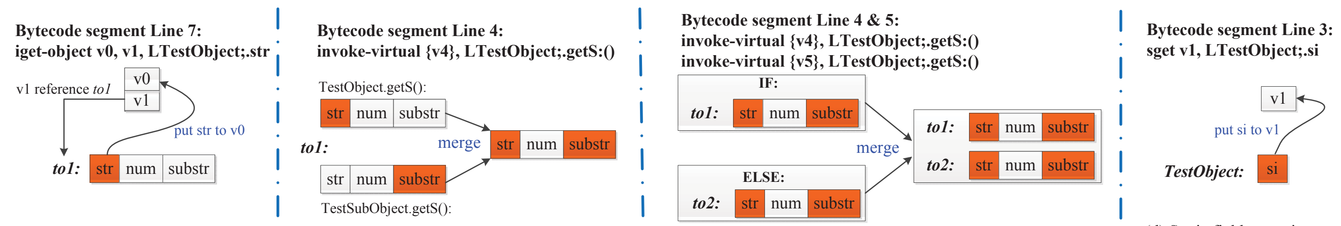
Fig. 3. Possible code execution paths for *bar()*, orange one is the actual execution path

an example, the *if/else* statement between Line 9 and Line 11 leads to two possible execution paths: one invokes *to1.getS()* while the other invokes *to2.getS()*. Moreover, since the class *TestObject* is inherited by *TestSubObject*, the invocation of the method *getS()* also has two possibilities and the run-time types of *to1* and *to2* can be either *TestObject* or *TestSubObject*. In particular, when the method *bar()* is invoked, the actual code execution path corresponding to the code segment in Figure 1(a) is shown in Figure 3.

To address such challenge, we will parse the bytecode instructions along each possible code execution path in the application binary to find out the relevant memory contexts that need to be migrated during workload offloading, and merge the parsing results of all these paths afterwards. Since the Dalvik VM uses a register-based architecture, the execution of bytecode instructions will affect the states of registers in the stack frame, and the method arguments are usually located in the last several positions of the register list. As a result, our parsing component emulates the effect of each bytecode instruction, tracks the register state, and records which fields of these arguments are operated accordingly. In Android, there are totally 217 types of bytecode instructions [19] that may appear in the application binary, and we describe the details of how we handle the few most common types of bytecode instructions as follows.

1) Object manipulation: Such instructions correspond to the bytecode *iget* and *iput*, and are the most commonly used in Android applications. *iget* means to get the value of an object field to the destination register. As shown in Figure 4(a), if the object (*to1*) operated by this bytecode instruction is from the method input arguments or their fields, we mark the corresponding field (*str*) of this object as to be migrated, and put this field into the destination register ($v0$), indicating that the subsequent operation to $v0$ equals to the operation to *str*. On the other hand, *iput* means to put the destination register into an object field, and further access of this field will return the same content as the destination register.

2) Method invocation: A method can be invoked by the bytecode instructions *invoke-kind*, such as *invoke-interface* and *invoke-virtual*. First, since a method may invoke some other methods, we need to recursively parse each method being invoked. Second, since the invocation of an interface method or a virtual method may be overridden by subclasses in Java, all the implementing classes of that interface and all the subclasses of the class which defines the virtual method need to be parsed, no matter whether they reside in the application binary or the OS kernel. The parsing results over



(a) Object manipulation (b) Method invocation (c) Branch instruction (d) Static field operation

Fig. 4. Handling the bytecode instructions with respect to Figure 1(b)

these implementing classes or subclasses are then merged to ensure that all the possible application execution paths at run-time can be covered. For example in Figure 4(b), the bytecode instruction will lead to parsing of both *TestObject.getS()* and *TestSubObject.getS()*. As a result, both fields *str* and *substr* of *to1* are marked as the contexts to be migrated.

3) Branch instructions: Such instructions correspond to the bytecode *switch* and *if-test*, and generate new branches of code execution paths. When these instructions are encountered, the parsing process will be split for each possible code execution path. For example in Figure 4(c), the *if* instruction will mark the operations on *to1*, while the *else* instruction will record how *to2* is operated. By combining the results, we end up with that the fields *str* and *substr* of both *to1* and *to2* may be accessed during method execution.

4) Array operation: Such instructions correspond to the bytecode *aget* and *aput*. Operation to array is a special case since the element to be operated can be only determined by the register contents at run-time. Therefore, our offline parsing has to mark all the elements in the array as to be migrated.

5) Parsing end: The instruction *return* indicates the end of the current method. If this method is not invoked by any other method, the current parsing path terminates and the parsing result is merged with that of other paths.

B. Class Static Field Parsing Component

This component aims to find out which classes and their static fields may be accessed during the execution of an application method. Our basic approach of such parsing is also to screen the application binary and parse the bytecode instructions. In particular, operations on class static fields correspond to the instructions *sget* and *sput*, which indicate getting or setting the value of a class static field to or from a register. Since writing a value to a field does not require the original value of this field to be correct, the appearance of *sput* instruction will be ignored. For the *sget* instruction, its operand indicates the class static field that it operates on. As a result, our parsing component resolves the class static field and records this static field as to be migrated during workload offloading. For example in Figure 4(d), the bytecode instruction allows the parser to mark the static field *si* of class *TestObject* as to be migrated.

Being different from the Method Argument Parsing Component, the parsing of class static fields can be done without taking the diversity of code execution paths into consideration, because the register status is not required for resolving the reading operations over static fields. Instead, we only need to scan the instructions defined in the method being parsed and all the recursive method invocations. Take the invocation of the method *bar()* in Figure 1(a) as an example, we only need to scan the binaries of *bar()*, *TestObject.getS()* and *TestSubObject.getS()*. As a result, it is found out that the static field *si* of class *TestObject* will be read when executing *bar()*.

C. Metadata Maintenance

The parsing results need to be efficiently recorded and maintained so that they can be applied for run-time migration and selectively migrating the relevant memory contexts for remote method execution. In practice, the memory contexts of a Java class object can be organized as a tree-based structure, with the object itself as the root. All the instance fields and static fields of a class object can be considered as the children of the root object. These fields can have their own children if they are reference type. This tree structure may continue recursively until a field is a primitive type or a reference type without any member fields. Such a field then becomes a leaf of the tree. As a result, we are able to maintain the parsing results based on breadth-first search of the object trees.

TABLE I
FORMAT OF METADATA

Result	Comment
LSample; bar 1	Method key
1|101	*str* and *substr* of *to1* need to be migrated, while *num* will not be accessed at run-time
1|101	The memory context of *to2*, which is similar to that of *to1*
LTestObject;	Class whose static field may be read
1	Static *si* of *TestObject* which needs to be migrated

To record the parsing results to the metadata file, we generate a unique key for each method first. The string combination of the name of the class which defines the method, the method name, and the method index generated by *dexgen* is used as the unique key for indexing. For every method argument object, we use breadth-first search to traverse its tree structure. If its child field will be accessed at run-time, "1" will be written into the metadata file, otherwise "0" is written. An "|" delimiter is used to indicate the end of listing the memory contexts of an object. The class field parsing result will be written into file after all the arguments parsing results have been recorded. For each class which may be accessed in the method, we will output its class name and the list of its static fields, with "1" or "0" to indicate that this field will be accessed or not.

For example in Figure 1(a), the memory contexts for *bar()* after parsing is shown in Fig 5, and the format of the generated metadata from parsing the method *bar()* is described in Table I. All the metadata for one application will be stored into a single file, which makes the file very large. Based on our observation, most spaces in the metadata file are taken by the full names

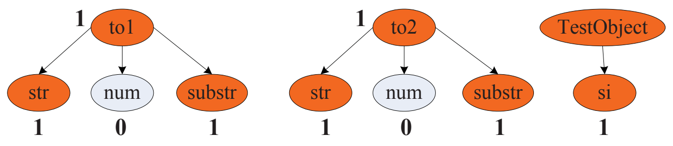


Fig. 5. The memory context for *bar() after parsing*

of classes which may be accessed in method, and these full names usually share much in common with each other due to the Java package hierarchy. For example, *android.os* is almost used as the prefix for any class related to basic operating system services. Therefore, to reduce the metadata file sizes, we replace all the character strings of class names to a unique numeric ID, and maintain an indexing dictionary to decode the ID at run-time. To provide an time-efficient file indexing mechanism, instead of loading the whole metadata file into the memory at run-time, we write another file to record the offset of each method in the metadata file. Since it is only one line for a method, the size of the offset file will be small enough to load into mobile memory during application starts. At run-time, the loaded offsets will be used to look up the metadata in metadata file.

V. Run-time Migration

Our Run-time Migration component monitors the run-time execution of Android applications, and supports remote method executions by utilizing the offline parsing results and migrating the relevant memory contexts to the remote cloud. Such migration process consists of four major steps: i) method invocation tracking, ii) context migration to the cloud, iii) context reload on the cloud, and iv) backward context migration to local device. Our implementation of these steps is integrated with the Dalvik VM in Android and involves about 2,000 lines of code in C.

A. Method Invocation Tracking

To support workload offloading at the level of different application methods, the invocation of each application method in an executing thread must be identified and recorded so that the application profiler can be launched and the offloading operations can be performed at the entry and completion point of the method. In general, there are two ways of method invocations in Android. One is invoked from a native method with the entry point of *dvmInterpret* in code. The other is from the invocation by a Java method through bytecode instructions *invoke-kind*. Both types of entry points are tracked by our system at run-time. If a method is going to be offloaded, our system will intercept the method invocation and migrate the relevant memory contexts to the cloud.

B. Context Migration to the Cloud

Our run-time migration component aims to collect and migrate the least but sufficient memory contexts to ensure remote method execution on the cloud. First, we will only migrate the stack frame of the corresponding method to be offloaded, rather than any other stack frames in the executing thread. Second, there are only two types of memory contexts that are accessible to a method, i.e., the method arguments and class static fields. Since the method arguments are located in

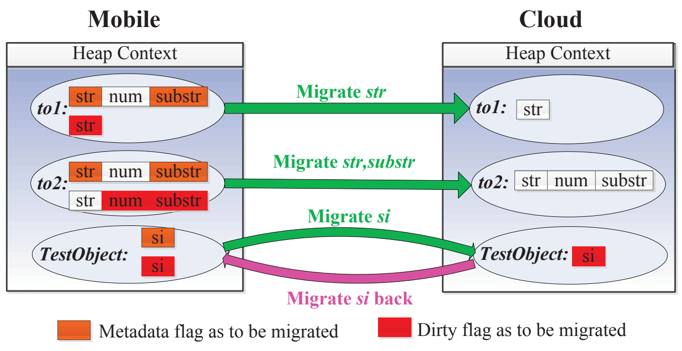


Fig. 6. VM synchronization during context migration

the last several positions of the register list in the stack frame, we can collect all the arguments contexts by resolving the method stack frame. If an argument is a primitive type, its value in register will be collected directly. If it is a reference type, the metadata generated during the offline parsing will be applied to collect the appropriate memory contexts in the memory heap. It will recursively traverse all the fields of the argument and check if the field needs to be migrated.

However, in practice we may find that the metadata records a larger number of fields from the actual amount at run-time, because our metadata is generated by combining the parsing results of all the possible application execution paths. For example in Figure 6 which corresponds to the invocation of method *bar()* in Figure 1(a), the run-time type of the first argument *to1* of *bar()* has only one instance field, while the metadata indicates it has three fields because the metadata combines the application execution paths of both *TestObject* and *TestSubObject*. In this case, we just omit the second and the third field indicated in the metadata, and only migrate the field *str* to the remote cloud.

Meanwhile, we further reduce the amount of data traffic for context migration by applying dirty flags on object fields. A dirty flag indicates that the value of the corresponding field is modified since last migration of this field, and hence needs to be migrated to ensure that cloud gets the latest value during operation. The field which is not flagged as dirty, on the other hand, should be avoided to be migrated since the cloud already has the latest copy. Thus for the *to2* in the Figure 6, even though the metadata indicates the fields *str* and *substr* of *to2* needs migration, the applying of dirty bits makes the offloading migrate only field *substr* of *to2*.

On the other hand, the migration of class static fields is similar to the migration of method arguments. The metadata maintains a list of class names which may be read when the method executes. We first test if a class on the list has been loaded into VM. If not, we can skip the migration of this class because the cloud VM will load this class when the method needs to use it. Otherwise, the contexts of the fields of this class will be collected recursively with metadata and dirty flags. We adopt the same technique being used in COMET [8] to solve the problem of reference addressing between two endpoints in the executing thread, by assigning an ID to each object during migration.

C. *Context Reload on the Cloud*

In the remote cloud, a Dalvik VM instance complied for the x86 architecture is launched to execute the offloading method. When there is an offloading event from a local mobile device, the cloud VM will receive all the data transmitted from the local device and parse them into its own run-time context. It's a reversed process of the context migration performed on the local mobile device. The cloud VM will first deserialize the contexts and then merge these contexts into its heap space. After that, a stack frame will be created for this offloading method into the thread and the thread starts to run.

D. *Context Migration Back to Local Device*

To support such backward context migration, we develop a specialized scheme to collect the memory contexts at the remote cloud after the completion of remote method execution, and migrate these contexts back to the local mobile device. We track all the memory objects in the executing thread of the cloud VM to be aware of all the objects being modified during the remote method execution. When migrating the memory contexts back to the local device, we migrate all the dirty fields of these objects to assure that the memory contexts in local device's memory heap are identical to that on the remote cloud. For example in Figure 6, the remote execution of *bar()* modifies *TestObject.si*, which is marked as dirty and will be migrated back to the local device.

We consider the following three types of scenarios, where a method being executed at the remote cloud needs to be migrated back to the local device.

- *Method return*: When the method finishes its execution on cloud, it needs to be migrated back to the local device. In this scenario, along with the dirty objects, the return value of method execution is also required to be migrated back. However, all the other local variables in the stack frame will not be used any more because these variables are only valid within the scope of the method being executed remotely. These contexts will not be migrated back and the corresponding data transmission cost is saved.
- *Exception throw*: A method may throw an exception during its execution. We will first try to see if this exception can be caught within method. If so, the method can continue to run; otherwise, this exception needs to be propagated to its invoker. The offloaded method being executed at the remote cloud, however, has no idea about its caller, because only the stack frame of this method is migrated. In this case, the handling of this exception must be done at the local device, and hence we are forced to migrate this method execution back to the local device. We will bypass the migration of all the local variables and only migrate all the dirty objects.
- *Native method*: When the offloading method invokes a native method which cannot be executed in the remote cloud due to the involvement of specialized hardware-related instructions or local resource access, it needs to be migrated back to the local device to ensure the smooth execution of the application. This migration, being different from the two cases above, requires all the local variables of the remotely executed method to be migrated back. In this case, we will go through all the memory contexts in the stack frames of the executing thread at the remote cloud, and migrate them all together with all the dirty objects.

VI. Performance Evaluations

In this section, we evaluate the effectiveness of our workload offloading system on reducing the local devices' resource consumption over various realistic smartphone applications. The following metrics are used in our evaluations:

- **Method execution time**: The average elapsed time of method executions over multiple experiment runs.
- **Energy consumption**: The average amount of local energy consumption over multiple experiment runs.
- **Amount of data transmission**: The average amount of data transmission during offloading over multiple experiment runs.

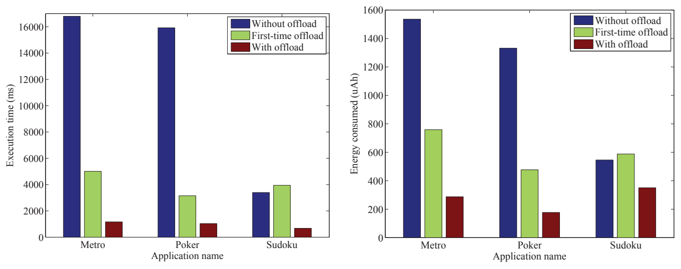
A. *Experiment setup*

Our experiments are running on Samsung Nexus S smartphones with Android v4.1.2, and a Dell OptiPlex 9010 PC with an Intel i5-3475s@2.9GHz CPU and 8GB RAM as the cloud server. The smartphones are connected to the cloud server via 100Mbps campus WiFi. We use a Monsoon power monitor to gather the real-time information about the smartphones' energy consumption. We evaluated our system with the following Android applications that are available on the Google Play App Store. The binaries of each application is first applied to our Offline Parsing Component, and then executed by our offloading system. Each experiment on a mobile application is conducted 30 times with different input datasets for statistical convergence.

- **Metro**: A trip planner which searches route between metro stations using Dijkstra's Algorithm[1].
- **Poker**: A game assisting application which calculates the odds of winning a poker game with Monte Carlo simulation[2].
- **Sudoku**: A sudoku game which generates a game and finds a solution[3].

As stated before, our major focus of this paper is to develop systematic techniques which improve the efficiency of context migration to the remote cloud. Therefore, we do not focus on addressing the problem of *what to offload* and determining the appropriate set of application methods to be offloaded. Instead, in our experiments, we follow the same methodology being used in [8] and use the historic records of the method execution times as the criteria for offloading an application method. More specifically, at the initial stage of each experiment, we let a method run locally for 30 times and calculate its average execution time. If such average execution time exceeds a given threshold, this method will be offloaded for remote execution in the future. We dynamically update this threshold at run-time according to the application executions.

[1]https://play.google.com/store/apps/details?id=com.mechsoft.ru.metro
[2]https://play.google.com/store/apps/details?id=com.leslie.cjpokeroddscalculator
[3]https://play.google.com/store/apps/details?id=com.icenta.sudoku.ui



(a) Method execution time (b) Energy consumption

Fig. 7. Performance evaluation on execution time and energy consumption

As we discussed in Section IV-A, the variety of code execution paths grows exponentially with the number of program branches and the frequency of class inheritance, and hence may either deplete the parsing server's local memory or increase the time of offline parsing. Therefore, in our experiments we make a tradeoff between the completeness of offline parsing results and the parsing overhead. More specifically, we empirically limit the parsing depth over program branches to 10 and the parsing depth over class inheritance to 15. When either a program branch or class inheritance in practice exceeds such limit, we will mark all the arguments of the corresponding method as to be migrated. We may further improve our algorithm in the future to relax such limits and reduce the parsing overhead without impairing its accuracy.

B. Effectiveness of Workload Offloading

We first compare the method execution time between local and remote executions. From the experimental results shown in Figure 7(a), we can see that we can achieve a remarkable speedup in method execution by offloading the methods to remote execution. For the Metro and Poker applications, we can reduce 90% of their execution time. For the Sudoku application with a shorter execution time, we can still achieve roughly 5 times speedup. In particular, the case of "First-time offload" in the figure means the first time when the application methods are offloaded to the remote cloud. This is a special case since a large set of class static fields that will never be changed in later execution needs to be migrated and hence incurs additional execution time.

TABLE II
AMOUNT OF DATA TRANSMISSION DURING WORKLOAD OFFLOADING

Appli-cation	First-time offload (KB)		Upload (KB)		Download (KB)	
	Ours	COMET	Ours	COMET	Ours	COMET
Metro	5,175.6	7,623.3	99.4	937.3	9.8	31.4
Poker	3,223.8	5,674.4	17.2	64.2	1.2	13.7
Sudoku	4,925.4	6,644.3	61.5	201.9	45.9	16.4

Meanwhile, the local energy consumption is significantly reduced as well by offloading. As shown in Figure 7(b), the intensive computations for the Metro and Poker application will consume a lot of local battery energy. With workload offloading, we can reduce more than 80% of local energy consumption. Comparatively, the energy saving for Sudoku is lower, which is about 35% since its computational complexity is less than the other two applications. Being similar with the cases of method execution times, the energy consumption for the first-time offloading is also higher than further offloading operations, due to the one-time migration of the class static fields.

We also evaluated the amount of data transmission during workload offloading, by comparing our proposed offloading system with COMET [8]. The evaluation results are listed in Table II. In general, we can achieve notable data transmission reduction. For the first-time offloading in each application, we can save the data traffic around 40% by screening out the class static fields which will not be used in this offloaded method. In particular, for the amount of upstream data transmission, we can reduce nearly 90% of the data transmission in Metro. Even for the worst case in Sudoku, the decrease of data transmission in our system can still reach up to 70%. The major reason for such advantage is that our scheme is able to predict which contexts will be used during method execution and hence selectively migrates them. For the downstream data transmission, our scheme normally transfers less data with an exception in Sudoku. By analyzing the offloaded methods, we find that such additional data transmission is incurred by the offloading decision criteria we adopted. Our decision criteria leads to offloading the instantiation of the large Puzzle object, which is migrated back to the local device afterwards. This case can be avoided by applying more online application profiling information on offloading decisions.

TABLE III
OFFLINE PARSING TIME AND METADATA FILE SIZE

Appli-cation	Parsing time (s)		File size (KB)	
	Method argument	Static field	Metadata file	Offset file
Metro	2	98	696.9	6.3
Poker	1	96	60.6	0.9
Sudoku	866	103	4,160.8	73.1

C. Parsing complexity

In this section, we evaluate the parsing time and size of metadata of our offline parsing component proposed in Section IV. As shown in Table III, we are able to generally control the offline parsing time within two minutes, which ensures prompt response to the subsequent user operations on mobile applications after installation. One exception is noticed on the Sudoku application which may take up to 10 minutes to be parsed and lead to a metadata file with a size larger than 4MB, because of its higher computational complexity and involvements of complicated program logic.

We also investigated the impact of parsing depth on the completeness of method argument parsing results on the Sudoku application. As shown in Figure 8, the larger threshold we set for the depth limit of parsing the program branches, the longer time the offline parsing will take and the more methods can be accurately parsed. With a branch depth limit as 10, we can parse 98.9% of methods encountered during parsing, but the parsing process may take up to 800 secs. When we reduce such limit down to 4, the percentage of application methods being parsed is only slightly reduced to 91.8%, with significant reduction of the parsing time down to 3 secs. We plan to further investigate the impact of such parsing depth on the reliability of remote method execution, and to develop adaptively algorithms to flexibly adjust such parsing depth at run-time according to the specific application characteristics.

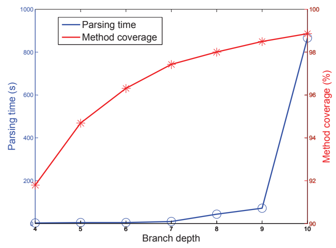


Fig. 8. Method argument parsing time and method coverage with different parsing thresholds

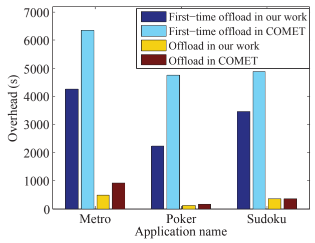


Fig. 9. Overhead in local mobile when offloading

D. *Offloading overhead*

In this section, we evaluate the computational overhead imposed during the run-time context migration, which is measured in the average amount of time spent on collecting the memory contexts to be migrated over all the offloading events throughout the application execution. We compare our offloading system with COMET [8]. By seeing the results shown in Fig 9, we can tell that the overhead in our system is 30% less than COMET [8] scheme during first offloading in application. The reason is that our scheme transfers much less data in first offloading. During further offloading, Metro can still get 35% less overhead, while the Poker and Sudoku have slightly less overhead than COMET [8] even though we save a significant amount of data transmission.

VII. Discussion

A. *Offline parsing*

In our proposed offloading system, we adopt an offline approach for parsing the application executables instead of an online approach, in order to reduce the run-time overhead of method migration. Our major motivation is that the application behavior at run-time is completely determined by its binary. As long as the application binary does not change, the possible variety of code execution paths and the memory contexts required by each execution path will remain the same as well. Therefore, each application method only needs to be parsed once offline. Another factor which drives us to use offline parsing is that we can take advantage of the strong computational power and large memory space in the remote cloud, where offline parsing is done. The limited computational resources at local mobile devices, on the other hand, cannot handle the parsing task because of the huge set of complicated class repositories in the OS kernel to be parsed.

B. *Multi-thread offloading support*

Our system supports multi-threads to be offloaded since every thread migrates enough contexts for itself to be executed smoothly at the remote cloud. During the execution of a method, the method may need to lock a memory object to synchronize with other threads. Since the memory contexts are shared between the local and cloud VMs via DSM, the synchronizing process needs to communicate to the other end-point of the thread execution to make sure that only one thread can lock on the object at anytime, and it takes much longer time than thread synchronization with only one VM. Thus, the performance of our system will decrease in applications which involves frequent synchronization among threads. In our future work, we will develop a better synchronization mechanism between thread endpoints to better support the current multi-threaded mobile applications.

VIII. Conclusions

In this paper, we presented a method-level offloading system which offloads the local computational workloads to the cloud with least context migration. Our basic idea is to use offline parsing to find the memory contexts which are necessary to method execution in advance and selectively migrate these contexts at run-time. Based on experiments over realistic mobile applications, we demonstrate that our offloading system can save a significant amount of energy while maintaining the same effectiveness of offloading.

References

[1] M. Satyanarayanan, “Mobile computing: the next decade,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 15, no. 2, pp. 2–10, 2011.

[2] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?” *IEEE Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[3] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, “Calling the cloud: Enabling mobile phones as interfaces to cloud applications,” in *Proceedings of the 10th International Middleware Conference*, 2009.

[4] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of ACM MobiSys*, 2010.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *Proceedings of INFOCOM*, 2012.

[6] W. Gao, Y. Li, H. Lu, T. Wang, and C. Liu, “On exploiting dynamic execution patterns for workload offloading in mobile cloud applications,” in *Proceedings of IEEE ICNP*, 2014.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[8] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, “Comet: Code offload by migrating execution transparently.” in *Proceedings of OSDI*, 2012, pp. 93–106.

[9] F. Hao, M. Kodialam, T. Lakshman, and S. Mukherjee, “Online allocation of virtual machines in a distributed cloud,” in *Proceedings of IEEE INFOCOM*. IEEE, 2014, pp. 10–18.

[10] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, “Cuckoo: a computation offloading framework for smartphones,” in *Mobile Computing, Applications, and Services*. Springer, 2012, pp. 59–79.

[11] L. Xiang, S. Ye, Y. Feng, B. Li, and B. Li, “Ready, set, go: Coalesced offloading from mobile devices to the cloud,” in *Proceedings of IEEE INFOCOM*. IEEE, 2014.

[12] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, “Odessa: enabling interactive perception applications on mobile devices,” in *Proceedings of MobiSys*, 2011.

[13] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards Wearable Cognitive Assistance,” in *Proceedings of ACM MobiSys*, 2014, pp. 68–81.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proceedings of USENIX NSDI*, 2005, pp. 273–286.

[15] F. R. J. Zhang and C. Lin, “Delay guaranteed live migration of virtual machines,” in *Proceedings of IEEE INFOCOM*. IEEE, 2014.

[16] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber, “Overview of distributed shared memory,” in *Technical Report, Trinity College Dublin*. Citeseer, 1998.

[17] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, “Diversity in Smartphone Usage,” in *Proceedings of MobiSys*. ACM, 2010, pp. 179–194.

[18] J. Huang, Q. Xu, B. Tiwana, Z. Mao, M. Zhang, and P. Bahl, “Anatomizing Application Performance Differences on Smartphones,” in *Proceedings of ACM MobiSys*, 2010, pp. 165–178.

[19] Android Developers, “Bytecode for the Dalvik VM,” https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.