Minimizing Context Migration in Mobile Code Offload

Yong Li, *Student Member, IEEE* and Wei Gao, *Member, IEEE*

Abstract—Mobile Cloud Computing (MCC) is of particular importance to address the conflict 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. To further improve the efficiency of such offline parsing of application binaries, our scheme also conducts one-time parsing to all the mobile OS libraries and reuses these parsing results for different user applications. We have implemented our design over the Dalvik Virtual Machine of Android OS. Our experiments and evaluation against applications downloaded from Google Play show that our approach can save data transmission significantly comparing to existing schemes.

Index Terms—Mobile cloud computing, code offload, context migration, distributed shared memory, virtual machine

1 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.

First, the computational workloads at local mobile devices need 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 level and directly interacts with the intact application binaries. Existing offloading schemes, in contrast, either rely on the application developers' offline efforts to declare the sets of application methods to be offloaded [4], [5], or 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

• *The authors are with the Department of Electrical Engineering and Computer Science, University of Tennessee at Knoxville, 1520 Middle Drive, Knoxville, TN 37996. E-mail: yli118@vols.utk.edu, weigao@utk.edu.*

Manuscript received 19 Nov. 2015; revised 26 May 2016; accepted 16 June 2016. Date of publication 29 June 2016; date of current version 2 Mar. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2016.2586056

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. In order to further improve the efficiency of such offline parsing and avoid unnecessary redundancy during parsing, we also pre-parse all the OS libraries that may be invoked by mobile application methods and then reuse these parsing results for different user applications. We have implemented the proposed system design over practical Android OS, and the experimental results over realistic smartphone applications show that our system can migrate 70 percent 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 develop a systematic approach to thoroughly parse the mobile OS libraries that may be invoked by application methods, and significantly improve the efficiency of application parsing by reusing the OS parsing results.
- 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 2 reviews the existing work. Section 3.1 introduces the Android system background related to our offloading system. Section 3.2 describes the motivation for our work, and Section 3.3 presents our high-level system design. Sections 4 and 6 present the technical details of our proposed techniques of offline parsing and run-time migration. Section 7 evaluates the performance of our system. Section 8 discusses and Section 9 concludes the paper.

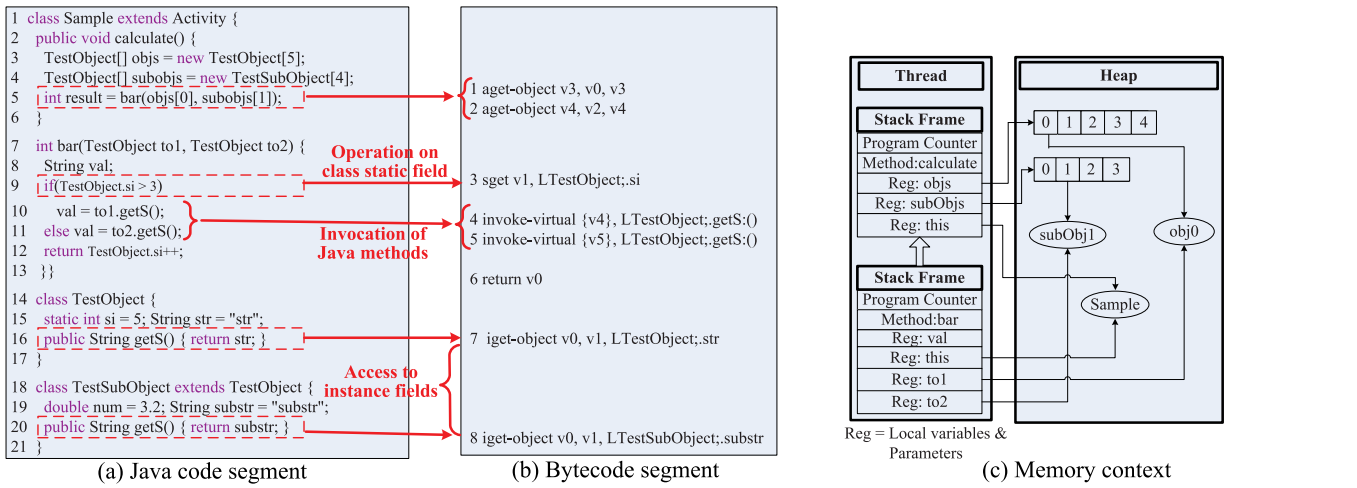
2 Related Work

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 [8], [10], [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. Some other approaches exploit the dynamic execution patterns of the applications to partition and schedule the workload offloading [6], [13], [14]. Nevertheless, our major focus 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], [15]. 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. Tango [16] flip-flops the leadership between VM replicas in the mobile device and back-end server, and then uses deterministic replay of inputs to ensure that different VM replicas perform the same computation. It hence can always get the same output at the mobile device without explicitly partitioning the programs.

CloneCloud [7] and COMET [8], being similar to our proposed system, offload the computational workloads through VM synthesis [17], [18]. 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) [19] 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.

Our proposed technique of offline program parsing is inspired by the existing work on static code analysis, which analyzes programs without actually executing them and is widely applied for compilation-time program optimization [20] and bug hunting [21]. Such techniques are also widely used over mobile platforms. For example, FlowDroid [22] analyzes the flow of taints in a mobile application to detect the attacks on users' privacy-sensitive data. eLens [23] estimates the code-level energy consumption of applications by combining static code analysis and per-instruction energy modeling. Our work, instead, applies static code analysis in the form of offline parsing to improve the efficiency of mobile offloading by migrating least contexts to the cloud, and hence has a completely different focus with the existing work.



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

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

3 Overview

3.1 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 [24], [25] 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 Fig. 1. This example will also be used throughout the rest of this paper to illustrate our ideas and system designs. As shown in Fig. 1b, 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 Fig. 1a). Second, operations on class static fields (e.g., *TestObject.si* in Line 9 of Fig. 1a) 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 Fig. 1a).

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 Fig. 1, when the method *bar()* is being executed, the thread stack and memory heap space is shown in Fig. 1c. The stack frame of *bar()* will point to 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.

3.2 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 Fig. 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.

3.3 Big Picture

In this section, we will describe our system architecture, as shown in Fig. 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 through an inter-procedural data flow analysis [26] on the mobile applications' java bytecodes. During the method execution, there are two possible sources of objects it can access. One is the input arguments and the other is the class

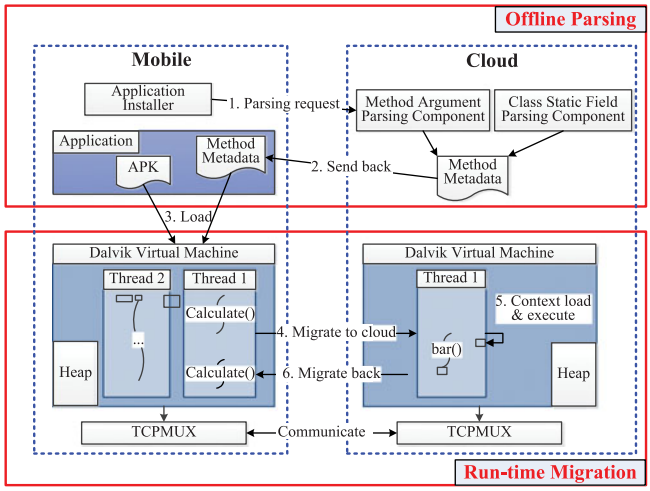


Fig. 2. The system architecture of the workload offloading system.

static fields. For example, in the method *bar()* of class *Sample* in Fig. 1a, 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 constructs a control-flow graph (CFG) for each application method, goes through all the possible execution paths in a CFG 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 Fig. 1a. 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.

4 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 emulating 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.

4.1 Method Argument Parsing Component

The major challenge of parsing the method arguments is the diversity of the possible application execution paths 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. To address such challenge, we will parse the bytecode instructions along all the possible code execution paths by constructing a CFG for each application method in the application binary to find out the relevant memory contexts that need to be migrated during workload offloading. A CFG is a graph representation of the possible execution paths for a method, where each node represents a specific instruction or instruction block and an edge is a code sequence flow. Such a node in a CFG, however, may need to be split in one of the following two cases.

First, a node is required to be split into branches representing possible branching paths when encountering a branch instruction, such as the *if/else* statement between Line 9 and Line 11 in Fig. 1a. Since we usually cannot know in advance the branch being actually executed at run-time, we have to construct the CFG in a conservative way by taking all possible branches into consideration.

Second, due to polymorphic invocations of an application method, a node in the CFG corresponding to that method may need to split, because the actual runtime type of the object which invokes the method is unknown during the parsing process. Thus a conservative way, which is to generate a node for each possible method being overridden, must be adopted.

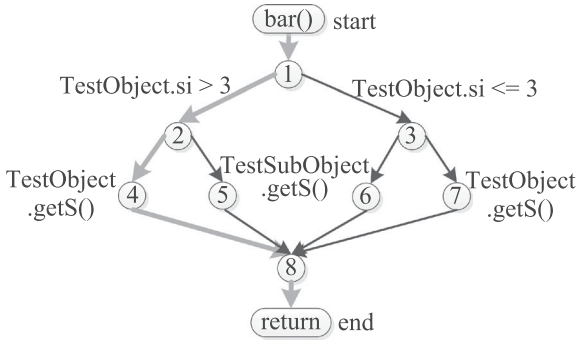


Fig. 3. Control-flow graph for *bar()*, orange one is the actual program execution path.

In particular, since the number of possible execution paths grows exponentially with the number of program branches and the number of child classes, the parsing component may consume too much system resources and take too long to finish. To address this issue, we define the parsing depth as the number of class inheritance branches and control the parsing flow based on it. With a higher parsing depth, more branches are constructed in CFG and more subclasses are included into the parsing process and hence more resources and time are required. Whenever the class inheritance in practice exceeds the threshold of parsing depth, the parsing component will skip the parsing of these methods and mark all the arguments of the corresponding method as to be migrated. For example in Fig. 1a, since the class *TestObject* is inherited by *TestSubObject*, the invocation of the method *getS()* has two possibilities and parsing depth of 2, 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 constructed CFG corresponding to the code segment in Fig. 1a is shown in Fig. 3.

Based on the above construction of CFGs, the parsing process traverses the constructed CFG from the start node to the end node with different actions. First, we analyze each node in the CFG to investigate the effects of the corresponding program instructions to the program data flow. Second, when a node splits into branches in the CFG, the parsing process splits as well to analyze the possible program execution path that may associate with each branch and different data flows. Finally, the parsing results over different branches in the CFG are merged. For example in Fig. 3, the parsing process starts at node 1, which then splits itself into node 2 and node 3 because of the *if/else* branching instruction. Node 2 will further be split into node 4 and node 5 because of the polymorphic method invocation to *to1.getS()*. Similarly, the process which handles node 3 will split into node 6 and node 7, and parse these different branches separately. After all the possible branches over the *if/else* instruction are parsed, the split parsing processes merge at node 8 and the parsing results in all the parsing processes are combined to get the final parsing result for the application method.

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 [27] 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.

4.1.1 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 Fig. 4a, 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.

4.1.2 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 libraries. The parsing results over 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 Fig. 4b, 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.

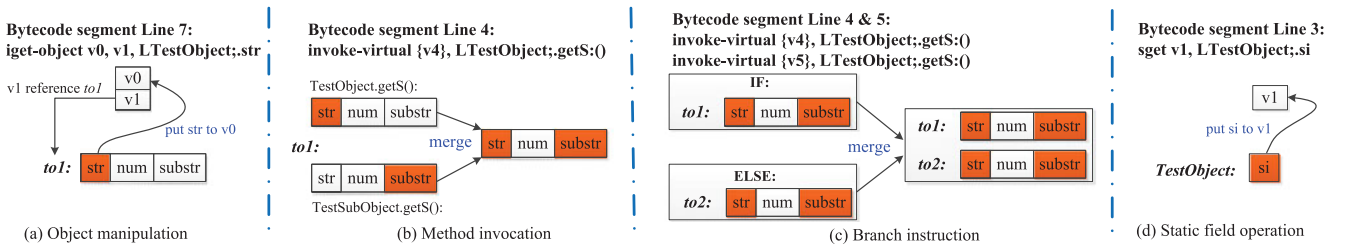


Fig. 4. Handling the bytecode instructions with respect to Fig. 1b.

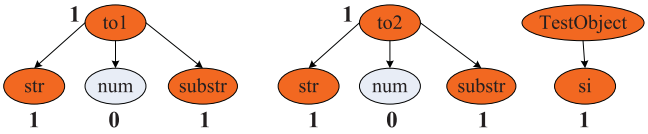


Fig. 5. The memory context for *bar() after parsing*.

4.1.3 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 in CFG for each possible code execution path. For example in Fig. 4c, 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.

4.1.4 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.

4.1.5 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.

4.2 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 Fig. 4d, 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 Fig. 1a 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()*.

TABLE 1
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

4.3 Metadata Maintenance

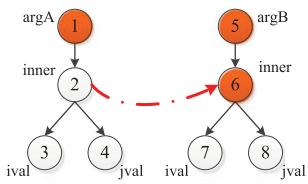
The parsing results need to be efficiently recorded and maintained so that they can be applied for run-time migration and selectively migrating the relevant memory contexts for remote method execution. In practice, the memory contexts of a Java class object can be organized as a tree-based structure, 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.

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 Fig. 1a, 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 1. 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 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

```
1 public class SysClass {
2   public static void innerB2A
    (SysObj argA, SysObj argB) {
3     argA.inner = argB.inner;
4   }}
5 class InnerObject {  int ival, jval;  }
6 class SysObj {  InnerObject inner;  }
7 class Sample extends Activity {
8   int bar2(SysObj so1, SysObj so2) {
9     SysClass.innerB2A(so1, so2);
10    return so1.inner.ival;
11 }}
```

(a) Java code segment



(b) Parsing result tree

Fig. 6. The object tree for arguments of method *SysClass.innerB2A()*.

method, the size of the offset file will be small enough to load into mobile memory during application starts. At runtime, the loaded offsets will be used to look up the metadata in metadata file.

5 OS Library Parsing

The Android OS has a multi-layered architecture and provides a large variety of APIs, in form of its software development kit (SDK), to allow user applications to interact with the OS, access system resources, and utilize the built-in system software tools for facilitating application functionalities. For example, Android SDK provides well-developed face detection algorithms to applications with the functionality of facial image processing, and also allows mobile applications to access system hardware resources through a collection of system services.

Mobile applications' invocation of these OS library methods may impair the efficiency of our offline parsing described above, because some of these OS library methods may take long time to be parsed. However, we notice most of these libraries are independent from the executions of mobile applications and have the same program behaviors during different application executions. Therefore, in this section, we present a set of special techniques to parse these OS libraries aside of the application parsing described above. Instead of parsing these OS libraries repetitively during application parsing, we propose to parse the OS libraries in advance and reuse such results when parsing different user applications, so as to improve the efficiency of offline parsing. The results of such OS library parsing are also persisted as metadata file. When an OS library method is found to be invoked by an application method, the corresponding OS library parsing result is loaded and then applied to the current process of application parsing.

5.1 Metadata Management for OS Library Parsing

Due to the change of program states during the execution of OS library methods, special techniques are needed to maintain and manage the metadata of the parsing results of OS libraries. Such change of program states happens when an instruction assigns a new value to a field of method arguments or a class static field. In this case, the process of application parsing should continue with the new program state after reusing the OS library parsing results. Therefore, the metadata for OS library parsing needs to record the effects of such assignments by mapping the field and its target object after the execution of OS library methods, so that such change of program states can be correctly retrieved and merged during user application parsing.

TABLE 2
Format of Metadata for *SysClass.innerB2A()*

Object ID	Field ID after OS library method	Access Info (1 means reading field at this offset)
1	6	0
...		
5	6	1
6	7, 8	0, 0
...		

5.1.1 Metadata Persistence

When parsing mobile applications, the metadata needs only to record the information concerning which fields of method arguments and which class static fields may be read throughout the method invocation, so as to retrieve the relevant memory contexts during run-time method offloading. However, the metadata for OS library parsing needs to contain additional information about the "*side effect*" of an OS library method, which occurs when the method modifies the state of the method arguments and the class static fields by assignment. Such information is the key to the correctness of application parsing when reusing the results of OS library parsing: after an assignment to a field in an OS library method, further operations to such an assigned field should be forwarded to the actual target object referenced by this field. Considering the example in Fig. 6a, after the execution of the OS library method *SysClass.innerB2A()* in Line 9, the field *inner* of argument *so1* will be referencing to the field *inner* of argument *so2*. This side effect by assignment must be recorded in the OS library parsing metadata, so that future reading to the field *ival* of argument *so1* in Line 10 will correctly mark the field *ival* of argument *so2*, instead of the field in argument *so1*, after applying OS library parsing results of *SysClass.innerB2A()* during parsing.

In order to store the side effect by assignment, the metadata needs to record the referencing target of each field. Our approach is to establish a mapping between the fields of method arguments or class static fields and their actual referencing target, after the execution of each OS library method. To establish such mapping, we use breath-first traversal to recursively assign an unique ID to each field of the method arguments and class static fields. For example in Fig. 6a, the method arguments and their recursive fields are assigned with IDs as shown in Fig. 6b. After the OS library method *SysClass.innerB2A()*, the field 2 is referencing to the field 6. Then the metadata persistence is to go through all the objects and each object writes out the object ID to which its fields are referencing, as well as the flags indicating if its fields have been accessed in the OS library method. The persistence of the metadata in the example is shown in Table 2. From the table we can see that the field *inner* of argument *argA* is recorded with ID 6, which is the same as the field *inner* of argument *argB*.

The parsing results of OS libraries will be written into a separate metadata file. In order to reduce the size of the parsing results, techniques such as indexing dictionary and offset files that were introduced in Section 4.3 will also be applied.

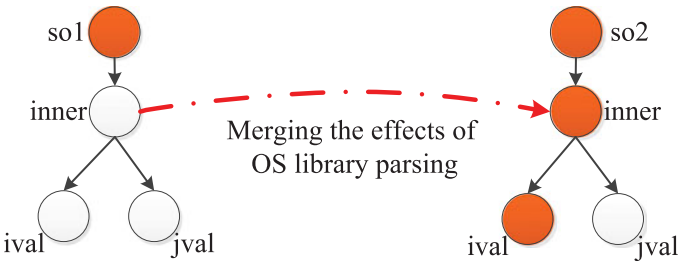


Fig. 7. Effects after merging OS library parsing results of *SysClass.innerB2A()*. Colored means the object may be accessed.

5.1.2 Metadata Retrieval

In order to reuse the OS library parsing results when parsing mobile applications, these parsing results must be first retrieved from the metadata file in a structured manner, and such retrieval process is a reverse counterpart of metadata persistence. First, data is loaded from the metadata file with the start position and length information by indexing the offset file, which is the list of object IDs and its field ID list as shown in Table 2. Then, the parsing results are reconstructed by traversing the list of objects and setting the fields of each object with the objects in its field ID list. If an object with a given ID does not exist, a new object with this ID is created. For example in Table 2, the object with ID 1 will find the object with ID 6 and set this object as its field. As a result, the reconstruction of parsing results from metadata will end up with the parsing result tree as shown in Fig. 6b.

5.2 Utilization of OS Library Parsing Results

In order to reflect the effects of OS library method invocations and ensure the correctness of application parsing without recursively parsing the OS library methods, the retrieved parsing results of OS libraries must be applied and merged to the actual arguments during application parsing. First, we merge the object access information for the actual method arguments in mobile applications, according to the access information in OS library parsing results. The corresponding field in the actual method arguments is marked as being read, if this field in the OS library method arguments may be read in the method. Second, the side effect to each field assignment is merged and the field is referencing to a new target object, if the field is changed by an OS library method.

After applying the OS library parsing results, the corresponding OS library methods are removed from CFG and hence skipped from application parsing. For example in Fig. 6a, the actual arguments of the OS library method *SysClass.innerB2A()* are *so1* and *so2*, which correspond to the OS library method arguments *argA* and *argB*, respectively. Consequently, *so2.inner* is first marked as being read since the mark of *argB.inner* is in the OS library parsing results. Afterwards, the side effects of field assignments are also merged, such that the field *so1.inner* is set to reference to *so2.inner* because of the reference of *argA.inner* to *argB.inner*. As a result, we assure that both the read and write operations in OS library methods are correctly applied to the application parsing results. For example in Fig. 6a, any further read of *ival* at Line 10 will actually mark the field *ival* of *so2.inner*, just as shown in Fig. 7.

5.3 Overridden OS Library Methods

The behaviors of OS library methods may be affected by user applications due to polymorphism, such that an application class may extend an OS library class and override the OS library class methods. Such polymorphism allows the application class to be passed to OS libraries through its OS library superclass reference, enabling an OS library method to invoke an application class. In this case, the OS library methods have to be parsed again at run-time to assure correctness, because we cannot know the existence of the application's subclass during offline parsing of OS library methods and hence cannot investigate the corresponding OS library method that invokes the overridden method.

More specifically, our proposed approach maintains the information about the invocation relationship among OS library methods. Before parsing the methods of a user application, the parsing component scans the list of application classes which are inheriting OS library classes. Then, OS library methods that invoke these overridden OS library methods are removed from being reused. As a result, when the parsing process encounters any of such removed OS library methods during the offline application parsing, the method will be re-parsed, so that the parsing results can be correctly revised according to the case that the OS method is invoking the application's overridden method.

6 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.

6.1 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.

6.2 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

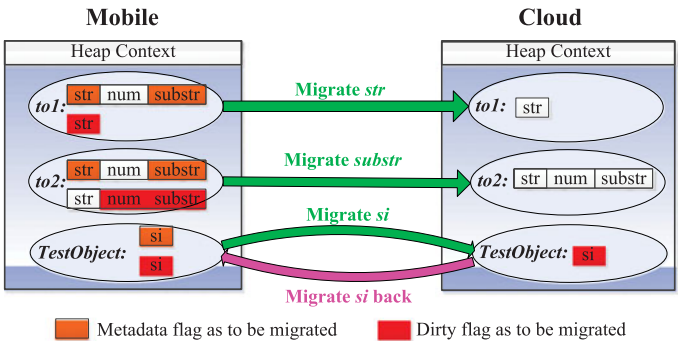


Fig. 8. VM synchronization during context migration.

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 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 Fig. 8 which corresponds to the invocation of method *bar()* in Fig. 1a, 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 Fig. 8, 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.

6.3 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.

6.4 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 Fig. 8, 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.

7 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.

We compare our scheme with COMET [8], the only existing scheme which also adopts a DSM-based approach. Other schemes [4], [5], [7] without DSM have to migrate the full run-time contexts required by the offloaded method, no matter if the memory objects have been synchronized before, leading to more context migration and data transmission than DSM-based schemes.

7.1 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@2.9 GHz CPU and 8 GB RAM as the cloud server. The smartphones are connected to the cloud server via 100 Mbps campus WiFi. Slower communication interfaces will further demonstrate the performance gain of our approach than other schemes since our approach migrates much less memory contexts over the wireless network link. 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. These applications cover the representative ones in the computational intensive categories. 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]

1. https://play.google.com/store/apps/details?id=com.mech-soft.ru.metro
2. https://play.google.com/store/apps/details?id=com.les-lie.cjpokeroddscalculator
3. https://play.google.com/store/apps/details?id=com.icen-ta.sudoku.ui

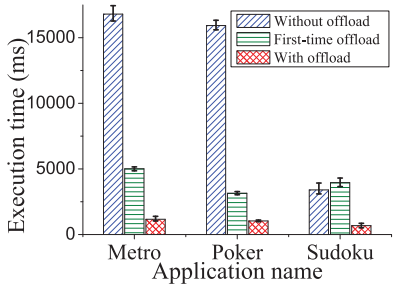


Fig. 9. Method execution time with/without offloading.

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 similar 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.

As we discussed in Section 4.1, 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. In practice, the program methods are not so complicated and hence have manageable program branches; while a few classes have a large repository of subclasses. 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 class inheritance to 15, unless explicitly mentioned in the paper. We may further improve our algorithm in the future to relax such limits and reduce the parsing overhead without impairing its accuracy.

7.2 Effectiveness of Workload Offloading

We first compare the method execution time between local and remote executions. From the experimental results shown in Fig. 9, 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 percent 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.

Meanwhile, the local energy consumption is significantly reduced as well by offloading. As shown in Fig. 10, the intensive computations for the Metro and Poker application

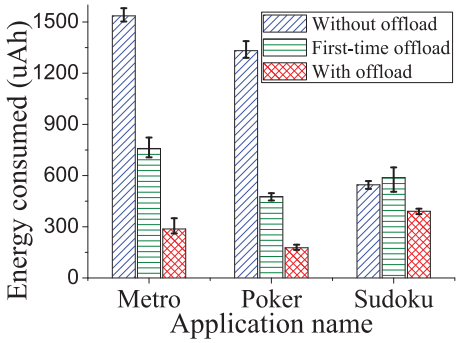


Fig. 10. Energy consumption with/without offloading.

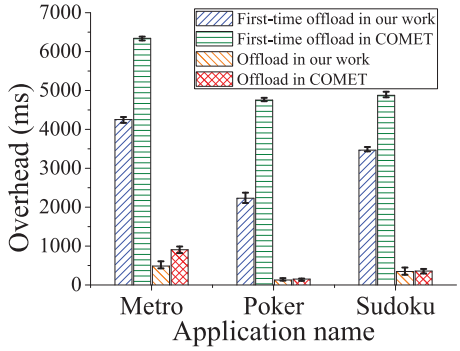


Fig. 11. Overhead on collecting and migrating the memory contexts.

will consume a lot of local battery energy. With workload offloading, we can reduce more than 80 percent of local energy consumption. Comparatively, the energy saving for Sudoku is lower, which is about 35 percent 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 3. 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 percent 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 percent 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 percent. 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; while the decision criteria in COMET [8] skip the offloading of this application method. This case can be avoided by applying more online application profiling information on offloading decisions.

7.3 Offloading Overhead

In this section, we evaluate the overhead imposed during the run-time context migration, which is measured in the average amount of time spent on collecting and migrating the memory contexts over all the offloading events throughout the application execution. This overhead is part of the execution time of the offloaded method in Fig. 9. We compare our offloading system with COMET [8]. By seeing the results shown in Fig. 11, we can tell that the overhead in our system is 30 percent less than COMET [8] scheme during the first offloading event in applications. The reason is that our scheme transfers much less data in first offloading. During further offloading, Metro can still get 35 percent less overhead, while the Poker and Sudoku have slightly less overhead than COMET [8] even though we save a significant amount of data transmission.

7.4 Parsing Complexity

In this section, we evaluate the parsing time and metadata size of our offline parsing component proposed in Section 4. As shown in Table 4, we are able to generally control the offline parsing time for applications within 13 seconds, 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 1 minute to be parsed and lead to a metadata file with a size larger than 4 MB, because of its higher computational complexity and involvements of complicated program logic. Furthermore, we notice that the parsing time for the OS library methods is more than 13 minutes and its metadata file exceeds 304 MB, because the Android OS libraries incorporate a large body of methods and takes more time to be parsed. However, this long parsing time of OS libraries will not affect the timeliness of application parsing because it is parsed in advance and is only an one-time effort. Even though the OS library metadata file cannot be all loaded into system memory due to its large size, its offset file is only around 3 MB, which can be all loaded into memory and indexed fast during application parsing.

TABLE 3
Amount of Data Transmission during Workload Offloading

Application	First-time (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

TABLE 4
Offline Parsing Time and Metadata File Size

Application	Method count	Parsing time (s)		File size (KB)	
		Argument	Static field	Metadata	Offset
Metro	391	5.6	6.9	696.9	6.3
Poker	59	5.3	5.5	60.6	0.9
Sudoku	15,504	66.7	50.6	4,160.8	73.1
Sys Lib	92,713	720.3	105.2	304,150.1	3,648.3

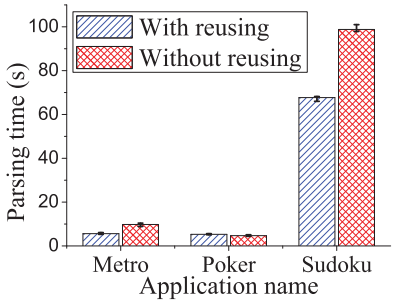


Fig. 12. Impact of reusing OS library parsing results on method argument parsing.

7.5 Efficiency of Utilizing OS Library Parsing Results

In this section, we evaluate the efficiency of reusing the OS library parsing results in offline application parsing, by measuring the average amount of time spent on application parsing. From the experimental results in Fig. 12, we can see that the application parsing time for method argument parsing component can have $1.5\times$ speed up in general. In particular, for the Poker application, the parsing time when reusing the OS library parsing results is slightly longer, because we need to index and load the metadata file and reconstruct the parsing result tree for the OS library methods, which takes longer time than parsing them. On the other hand, as shown in Fig. 13, the parsing time for class static field parsing component can also have at least $1.5\times$ speed up for all applications. In particular, the parsing time for Metro and Poker applications have more than $7\times$ speed up, due to the avoidance of repetitively parsing the OS library methods during application parsing. In particular, some OS library methods take long time to be parsed because the class static field parsing component needs to recursively scan all the invoked methods. Utilizing the pre-parsed metadata over these methods, therefore, could significantly reduce the time needed for application parsing.

7.6 Impact of Parsing Depth

In this section, we evaluate the impact of the parsing depth over class inheritance on the time of application parsing. As described in Section 4.1, with a higher parsing depth, more subclasses are included into the parsing process and hence more time is spent on parsing. From the experimental results shown in Fig. 14, we can see that the parsing time is increasing along with the parsing depth over class inheritance. We also note that the parsing time becomes stable and does not increase much after the parsing depth over class inheritance

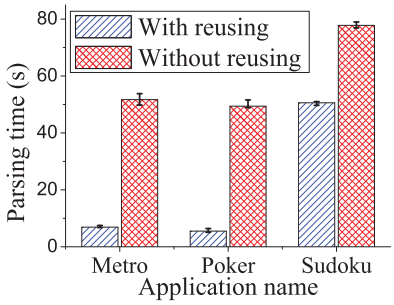


Fig. 13. Impact of reusing OS library parsing results on class static field parsing.

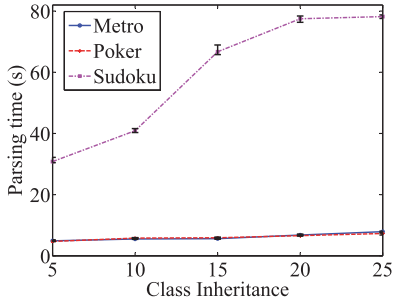


Fig. 14. Impact of parsing depth on parsing time.

reaches more than 20. This is reasonable since most classes in the system does not have a large amount of subclasses. Therefore, further increasing the parsing depth will not further involve more classes into the parsing process.

We also investigated the impact of the parsing depth on the completeness of method argument parsing results. As shown in Fig. 15, the higher parsing depth we use, the more methods can be accurately parsed, even though the longer time the offline parsing will take. For example on the Sudoku application, when the parsing depth is set to 25, we can parse 88.0 percent of application method but also increase the parsing time to 78.2 secs. When we reduce such depth down to 5, the percentage of application methods being parsed is only slightly reduced to 83.5 percent, with significant reduction of the parsing time down to 30.8 secs. We plan to further investigate the impact of such parsing depth on the efficiency of remote method execution, and to develop adaptive algorithms to flexibly adjust such parsing depth at run-time according to the specific application characteristics.

8 Discussion

8.1 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

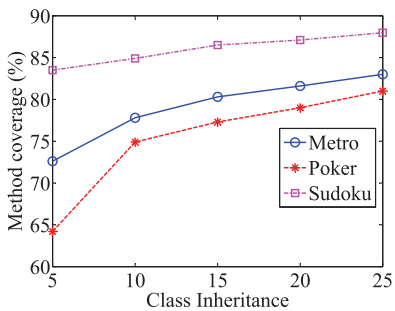


Fig. 15. Impact of parsing depth on method coverage.

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 libraries to be parsed.

8.2 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 endpoint 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, especially when the wireless connection is weak or lost. However, such degradation can be alleviated by incorporating robust offloading techniques similar as [28] or utilizing a private communication channel [29]. In our future work, we will develop a better synchronization mechanism between thread endpoints to better support the current multi-threaded mobile applications.

8.3 Supporting Other Mobile Platforms

Our proposed techniques of offline parsing and run-time migration can be applied to mobile platforms other than Android, as well, to reduce data traffic for computational workload offloading. For example, the Windows Mobile OS shares similar concepts with Android on application execution, which has a strong type system and uses a common language runtime (CLR) in the .NET Framework to interpret its bytecodes. Therefore, our proposed offline parsing technique can be performed on these bytecodes and the existing Windows mobile applications can be seamlessly supported by our proposed technique. On the other hand, the iOS platform is different from Android and Windows Mobile due to its binary application format, i.e., Mach-O, which makes it extremely difficult to do offline parsing on program binaries. We will leave the investigation of the possibility of offline program parsing over iOS applications as our future work.

8.4 Reusing the OS Parsing Results

Parsing the OS library methods in advance and reusing their parsing results to save the time of application parsing comes at a price of parsing accuracy degradation. More specifically, when the OS library methods are parsed alone, the information about the actual types of the method arguments received from user applications are lost. As a result, we have to adopt a conservative strategy to include all classes that are compatible with the declared type of method arguments. Another option, however, is to parse the OS library methods and user applications at the same time, so as to obtain the information about the actual type of the method arguments. For example in Fig. 1, when parsing only the method *bar()* itself, the actual types of *to1* and *to2* are missing. However, if its invoking method *calculate()* is taken into consideration, the parsing can at least make sure that the argument *to2* in *bar()* cannot be a *TestObject* because the actual argument passed to *to2* is declared as *TestSubObject* which is a child of *TestObject*.

9 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.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-1456656, CNS-1526769, and CNS-1527612. This work was also supported in part by the Army Research Office (ARO) under grant W911NF-15-1-0221.

References

[1] M. Satyanarayanan, "Mobile computing: The next decade," *ACM SIGMOBILE Mob. Comput. Commun. Rev.*, 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 Comput.*, vol. 43, no. 4, pp. 51–56, Apr. 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 *Proc. 10th Int. Middleware Conf.*, 2009, pp. 83–102.

[4] E. Cuervo, et al., "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mob. Syst. Appl. Serv.*, 2010, pp. 49–62.

[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 *Proc. Int. Conf. Comput. Commun.*, 2012, pp. 945–953.

[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 *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, 2014, pp. 1–12.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 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 *Proc. 10th USENIX Conf. Oper. Syst. Des. Implementation*, 2012, pp. 93–106.

[9] F. Hao, M. Kodialam, T. V. Lakshman, and S. Mukherjee, "Online allocation of virtual machines in a distributed cloud," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 10–18.

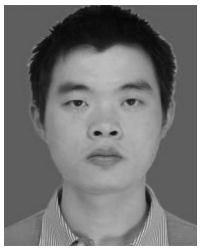
[10] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*. Berlin, Germany: Springer, 2012, pp. 59–79.

[11] L. Xiang, S. Ye, Y. Feng, B. Li, and B. Li, "Ready, set, go: Coalesced offloading from mobile devices to the cloud," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 2373–2381.

[12] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proc. 9th Int. Conf. Mob. Syst. Appl. Serv.*, 2011, pp. 43–56.

[13] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-efficient computation offloading in cellular networks," in *Proc. IEEE 23rd Int. Conf. Netw. Protocols*, 2015, pp. 145–155.

[14] L. Tong and W. Gao, "Application-aware traffic scheduling for workload offloading in mobile clouds," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1710–1718.
[15] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. 12th Annu. Int. Conf. Mob. Syst. Appl. Serv.*, 2014, pp. 68–81.
[16] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proc. 13th Annu. Int. Conf. Mob. Syst. Appl. Serv.*, 2015, pp. 137–150.
[17] C. Clark, et al., "Live migration of virtual machines," in *Proc. 2nd Conf. Symp. Netw. Syst. Des. Implementation*, 2005, pp. 273–286.
[18] F. R. J. Zhang and C. Lin, "Delay guaranteed live migration of virtual machines," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 574–582.
[19] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber, "Overview of distributed shared memory," in Trinity College, Dublin, Ireland, Tech. Rep. 895051, 1998.
[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. IEEE Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.
[21] A. Bessey, et al., "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[22] S. Arzt, et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, vol. 49, no. 6, 2014, pp. 259–269.
[23] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proc. IEEE Int. Conf. Softw. Eng.*, 2013, pp. 92–101.
[24] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proc. 8th Int. Conf. Mob. Syst. Appl. Serv.*, 2010, pp. 179–194.
[25] J. Huang, Q. Xu, B. Tiwana, Z. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proc. 8th Int. Conf. Mob. Syst. Appl. Serv.*, 2010, pp. 165–178.
[26] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 1995, pp. 49–61.
[27] Android Developers, Bytecode for the Dalvik VM. (2008). [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
[28] F. Berg, F. Durr, and K. Rothermel, "Increasing the efficiency and responsiveness of mobile applications with preemptable code offloading," in *Proc. IEEE Int. Conf. Mobile Serv.*, 2014, pp. 76–83.
[29] H. Lu and W. Gao, "Supporting real-time wireless traffic through a high-throughput side channel," in *Proc. 17th ACM Int. Symp. Mob. Ad Hoc Netw. Comput.*, 2016, pp. 311–320.



Yong Li (S'14) received the BE degree from East China Normal University in 2008, majoring in software engineering. Currently, he is working toward the PhD degree in the Department of Electrical Engineering and Computer Science, the University of Tennessee, Knoxville. His research interests include mobile cloud computing and software systems. He is a student member of the IEEE.

Wei Gao (M'05) received the BE degree in electrical engineering from the University of Science and Technology of China in 2005 and the PhD degree in computer science from Pennsylvania State University in 2012. He is currently an assistant professor in the Department of Electrical Engineering and Computer Science, the University of Tennessee, Knoxville. His research interests include wireless and mobile network systems, mobile social networks, cyber-physical systems, and pervasive and mobile computing. He received the US National Science Foundation (NSF) CAREER award in 2016. He is a member of the IEEE.