12-2018

Enhancing Mobile Capacity through Generic and Efficient Resource Sharing

Yong Li
University of Tennessee, yli118@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Yong Li entitled "Enhancing Mobile Capacity through Generic and Efficient Resource Sharing." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Wei Gao, Major Professor

We have read this dissertation and recommend its acceptance:

Michael R. Jantz, Hairong Qi, Wenjun Zhou

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Enhancing Mobile Capacity through Generic and Efficient Resource Sharing

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yong Li

December 2018

Abstract

Mobile computing devices are becoming indispensable in every aspect of human life, but diverse hardware limits make current mobile devices far from ideal for satisfying the performance requirements of modern mobile applications and being used anytime, anywhere. Mobile Cloud Computing (MCC) could be a viable solution to bypass these limits which enhances the mobile capacity through cooperative resource sharing, but is challenging due to the heterogeneity of mobile devices in both hardware and software aspects. Traditional schemes either restrict to share a specific type of hardware resource within individual applications, which requires tremendous reprogramming efforts; or disregard the runtime execution pattern and transmit too much unnecessary data, resulting in bandwidth and energy waste.

To address the aforementioned challenges, we present three novel designs of resource sharing frameworks which utilize the various system resources from a remote or personal cloud to enhance the mobile capacity in a generic and efficient manner. First, we propose a novel method-level offloading methodology to run the mobile computational workload on the remote cloud CPU. Minimized data transmission is achieved during such offloading by identifying and selectively migrating the memory contexts which are necessary to the method execution. Second, we present a systematic framework to maximize the mobile performance of graphics rendering with the remote cloud GPU, during which the redundant pixels across consecutive frames are reused to reduce the transmitted frame data. Last, we propose to exploit the unified mobile OS services and generically interconnect heterogeneous mobile devices towards a personal mobile cloud, which complement and flexibly share mobile peripherals (e.g., sensors, camera) with each other.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Mobile computing has been an indispensable part of modern life. Ideally, a mobile device should have powerful capabilities in computation, communication and sensing, an everlasting battery lifetime and a wearable form factor, so as to be used anytime and anywhere. However in reality, various hardware limits make it infeasible to realize this ideal objective, and tradeoffs must be made instead. For example, wearable devices have the small form factor at the cost of system performance and battery capacity. Further, mobile devices nowadays are designated to execute computationally expensive applications such as VR gaming, speech recognition, and video playback. The increasing complexity in these applications quickly exceeds the capability of any individual device and imposes a hard limit on the scalability of mobile computing system.

Mobile Cloud Computing (MCC) [71] could be a viable solution to bridge the gap between the limited capabilities of mobile devices and the increasing demands of user applications, by performing tasks with the shared resources from a remote or personal cloud. For example, smartphones can exploit the computational power of the cloud to improve their performance and save local battery, by offloading the expensive algorithm execution and high-quality graphics rendering to the cloud [22, 50, 75, 23, 52]. Meanwhile, they can provide their sensory data to other devices to facilitate the context-aware applications [41, 82]. However, the heterogeneity of mobile hardware and software system components severely hinders the generality and efficiency of such resource sharing in MCC.

Generality

The increasing variety of hardware components being mounted on today's mobile devices results in fundamental difference in the drivers, I/O stacks and data access interfaces being used by these hardware. Access to the same type of hardware resource from a remote system could fail if the hardware drivers are provided by different manufacturers and incompatible with each other. Moreover, the complexity of todays mobile applications has been dramatically increased with heterogeneous ways of accessing these resources. For example, infrequent retrieval of location information from GPS can be accomplished simply by method invocation; while access to the multimedia resources (e.g., speaker and camera) has to employ system shared memory to avoid the overhead of moving large sizes of bulk data.

Existing solutions, unfortunately, require manual efforts from application developers, which explicitly define how and when to access a specific type of shared resource from cloud [60, 86, 22, 50, 15]. Therefore, they will need a large amount of reprogramming efforts to access the various system resources remotely for individual applications or mobile devices, which severely impairs the usability of resource sharing in versatile environments. Instead, access to remote system resources needs to be performed automatically by the mobile Operating System (OS) without programmers' intervention, so that the large population of existing mobile applications can be supported without any modification or additional efforts of software redevelopment. To address this challenge, the remote resource access must be integrated within the OS level and directly interact with the intact application binaries.

Transmission Efficiency

Remote resource access entails frequent data exchanges between the mobile and cloud so as to provide necessary contexts for hardware operations and receive resource data from cloud. While in MCC, the mobile is usually connected with the cloud wirelessly through cellular or WiFi, which suffers from low bandwidth and high energy consumption. Therefore, the strategy of remote resource access needs to be adaptively adjusted according to the heterogeneous runtime execution patterns of individual applications, so as to maximize the profit of remote resource access with reduced transmission overhead. Existing solutions, however, largely ignore such runtime patterns and transmit too much unnecessary data, which results in bandwidth and energy waste. For instance, existing CPU offloading schemes [19, 35] recklessly migrate all memory contexts reachable from the current thread even though many of them will never be visited throughout the method execution. The GPU offloading schemes [52, 15] separately render and transmit every video frame to the mobile without fully exploiting the pixel redundancy across adjacent frames, resulting enormous transmission of frame data.

1.2 Motivation

In this section, we first briefly introduce the necessary background of the Android system, which is the most popular OS nowadays on various mobile platforms [30, 44] and our targeting system platform throughout this work. Due to the limited resources on mobile devices, mobile OSes usually adopt a hierarchical architecture as shown in Figure 1.1, so as to isolate user applications from low-layer system implementations. Such isolation leads to more efficient system resource management, and protects the mobile system from resource depletion due to poorly designed applications or malicious attacks from mobile malware. Mobile applications in such a hierarchical OS architecture do not access system resources directly. Instead, resource access is provided by unified mobile OS components. For example, the utilization of mobile CPU and GPU is achieved by executing the bytecode instructions in Dalvik VM and graphics commands in OpenGL ES respectively. While the interaction to system peripherals (e.g., GPS, speaker) is accomplished via individual system services (e.g., location service, audioflinger). Then, these mobile OS components interact with hardware device drivers through a hardware abstraction layer (HAL), whose interfaces are pre-defined by the OSes and implemented as libraries by the manufacturers.

Intuitive solutions to remote resource access [12, 56] would operate over low-layer device drivers directly, which however are ineffective due to the following reasons. First, the

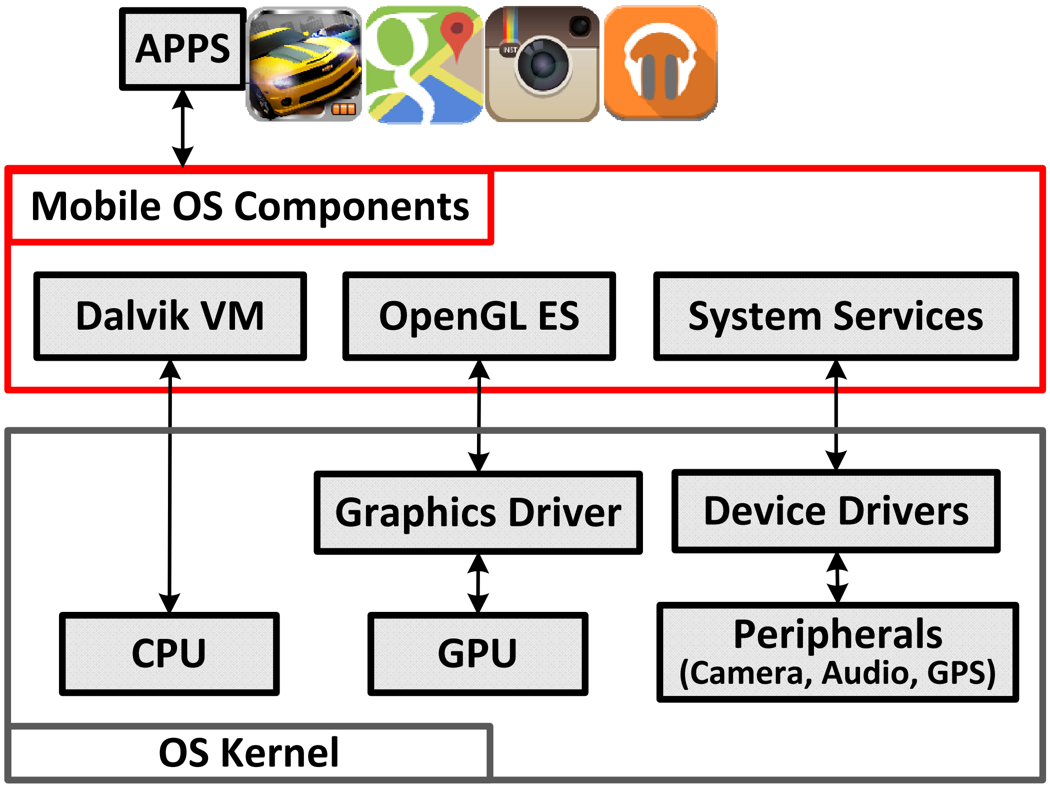


Figure 1.1: Layered architecture of Android mobile system

heterogeneity in device drivers creates discrepancy in the format and implementation details between the mobile and cloud, which incurs reprogramming efforts to make their drivers consistent for each type of hardware. Second, low-layer operations need to synchronize data between the mobile and cloud whenever the hardware status changes. Such synchronization scheme, however, is incapable of adapting to applications' actual resource requests, leading to large amounts of unnecessary data transmission. For example, each user application usually specifies its own interval of requesting for location data, despite that the GPS device in Android reports location information in every second.

Instead, we utilize the mobile OS components as the interface of remote resource access. Such unified OS components mask the heterogeneity in hardware operations and hence enable remote resource access in a generic manner. Therefore, mobile applications could speed up the execution by running the intact bytecode instructions in the cloud's Dalvik VM even though the cloud CPU may have a different instruction set architecture with the mobile CPU. Also, the mobile could improve its graphic quality and augment its ability of sensing the environment with cloud GPU and peripherals in spite of the heterogeneous hardware specifications.

On the other hand, we exploit the runtime characteristics of each hardware to reduce the data transmission during remote resource access.

- **CPU:** In order to offload mobile computational workload to remote cloud, the memory contexts in the stack and heap space have to be transmitted and restored in the cloud's Dalvik VM. However, the execution of an application method would only access a fraction of such memory contexts, which are limited to its own input arguments and global variables. Such application execution characteristics motivate us to reduce communication overhead by transmitting only the relevant memory contexts.

- **GPU:** Rendering remotely with cloud GPU necessitates to stream large frame data to the mobile. However, large amounts of redundant pixels are retained across adjacent frames, which could reach more than 90% [65]. Reusing such redundant pixels on mobile could effectively reduce the size of the frame being transmitted.

- **Peripherals:** In many cases, there is a gap between the production rate of the resource data and the needed rate by the applications. For example, a pedometer application samples the accelerometer in a much lower rate (10Hz [70]) than the reporting rate in hardware (100Hz). Therefore, we are motivated to regulate resource data transmission according to real-time application behaviors, which are specified during their interaction to the mobile OS components.

1.3 Contribution

1.3.1 A mobile offloading framework to access CPU on remote cloud

We present a novel design of offloading framework to exploit cloud CPU which performs automated method-level workload offloading in Dalvik VM with least context migration. The framework first identifies the memory contexts which may be accessed by a specific method during its execution through offline parsing to application binaries. Then the thread stack and heap contexts are screened at runtime to migrate only such relevant memory contexts to the remote cloud. The proposed framework is implemented over practical Android OS, and the experimental results over realistic smartphone applications show that the system can migrate 70% less memory contexts compared to existing schemes, while maintaining the same offloading effectiveness.

1.3.2 A mobile VR framework to access GPU on remote cloud

We present *DeltaVR*, a systematic mobile VR framework that utilizes the remote cloud GPU to render high-quality graphics and maximize mobile VR performance. Being different from traditional schemes which completely transmit every VR frame to the mobile, DeltaVR utilizes the cloud's computational power to explicitly decide the pixel redundancy across adjacent frames. Such pixel redundancy is then eliminated from the VR frame data and hence only the distinct portions of each frame would be transmitted to mobile devices. DeltaVR has been implemented over the Android OS and Unity VR application engine as a mobile middleware between VR applications and OS drivers, so as to ensure its generality over different VR applications with heterogeneous dynamics and computation demands. The experimental results over real-world VR applications show that DeltaVR can maximize the VR performance with complicated scenes, while reducing more than 95% of the VR frame data being wirelessly transmitted.

1.3.3 A framework to access mobile peripherals on personal cloud

We present a resource sharing framework that allows the mobile devices owned by a user to complement each other with generic access to system peripherals. The framework generically interconnects heterogeneous mobile devices towards a personal mobile cloud and shares peripherals within the cloud via remote access and invocation of the unified system services. The proposed framework is implemented as a middleware on Android OS over various mobile platforms with diverse characteristics and resource limits. The evaluation results show that the design can efficiently support ubiquitous access to peripheral resources between remote systems for arbitrary mobile applications, without incurring any significant system overhead.

1.4 Organization

The remainder of this proposal is organized as follows. Chapter 2 introduces a workload offloading framework to access cloud CPU with minimized context migration. Chapter 3 describes DeltaVR as a GPU offloading framework to achieve high-performance mobile VR with reduced frame data transmission. Chapter 4 presents a resource sharing framework to allow generic access to remote peripherals. Chapter 5 concludes the dissertation.

Chapter 2

Code Offload with Least Context Migration in the Mobile Cloud

2.1 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) [71] 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 [51]. Intensive research has been conducted on how to appropriately decide such application partitioning [33, 22, 50, 32], and support remote application execution through techniques of code migration [22, 50] and Virtual Machine (VM) synthesis [19, 35, 40]. However, these traditional schemes either restrict the scope of workload offloading to a specific set of system frameworks and mobile applications [22, 50], 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 [19, 35]. 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 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 [22, 50], and lack of the capability of run-time application partitioning and profiling [19].

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 [19, 35] 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 chapter, 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.

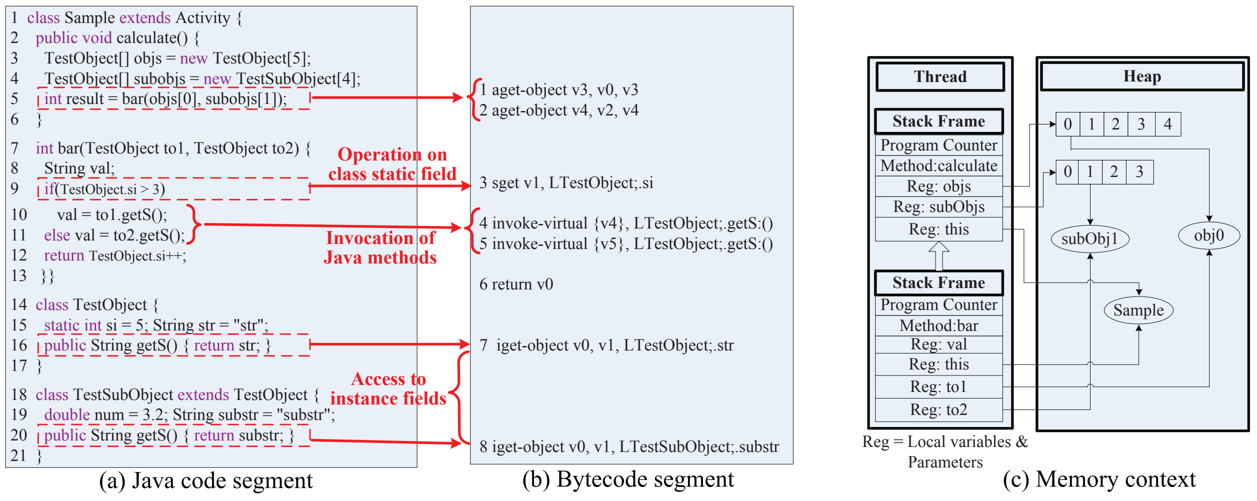
The rest of this chapter is organized as follows. Section 2.2 reviews the existing work. Section 2.3.1 introduces the Android system background related to our offloading system. Section 2.3.2 describes the motivation for our work, and Section 2.3.3 presents our high-level system design. Sections 2.4 and 2.5 present the technical details of our proposed techniques of offline parsing and run-time migration. Section 2.6 evaluates the performance of our system. Section 2.7 discusses and Section 2.8 concludes the chapter.

2.2 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 [22] and ThinkAir [50], 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 [49, 35, 85] 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 [68] assumes linear speedup over consecutive frames in a face recognition application. ThinkAir [50] defines multiple static offloading policies, each of which focuses on a sole aspect of system performance. Nevertheless, the major focus of this chapter 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 [71, 38]. MAUI [22] 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 [50] 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 [19] and COMET [35], being similar to our proposed system, offload the computational workloads through VM synthesis [20, 45]. CloneCloud [19] 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) [46] technique. Similar technique is also used in COMET [35], 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.



```
1  class Sample extends Activity {
2    public void calculate() {
3      TestObject[] objs = new TestObject[5];
4      TestObject[] subobjs = new TestSubObject[4];
5      int result = bar(objs[0], subobjs[1]);
6    }
7    int bar(TestObject to1, TestObject to2) {
8      String val;
9      if(TestObject.si > 3)
10       val = to1.getS();
11     else val = to2.getS();
12     return TestObject.si++;
13   }}
14 class TestObject {
15   static int si = 5; String str = "str";
16   public String getS() { return str; }
17 }
18 class TestSubObject extends TestObject {
19   double num = 3.2; String substr = "substr";
20   public String getS() { return substr; }
21 }
```

```
1 aget-object v3, v0, v3
2 aget-object v4, v2, v4
3 sget v1, LTestObject;.si
4 invoke-virtual {v4}, LTestObject;.getS:()
5 invoke-virtual {v5}, LTestObject;.getS:()
6 return v0
7  iget-object v0, v1, LTestObject;.str
8 iget-object v0, v1, LTestSubObject;.substr
```

Figure 2.1: An example of the execution model of Android applications

2.3 Overview

2.3.1 Background of Android System

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 2.1. This example will also be used throughout the rest of this chapter to illustrate our ideas and system designs. As shown in Figure 2.1(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 2.1(a)). Second, operations on class static fields (e.g., *TestObject.si* in Line 9 of Figure 2.1(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 2.1(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 2.1, when the method *bar()* is being executed, the thread stack and memory heap space is shown in Figure 2.1(c). 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.

2.3.2 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 Figure 2.1, when the method *bar()* is going to be offloaded for remote execution. In traditional offloading schemes such as COMET [35], 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.

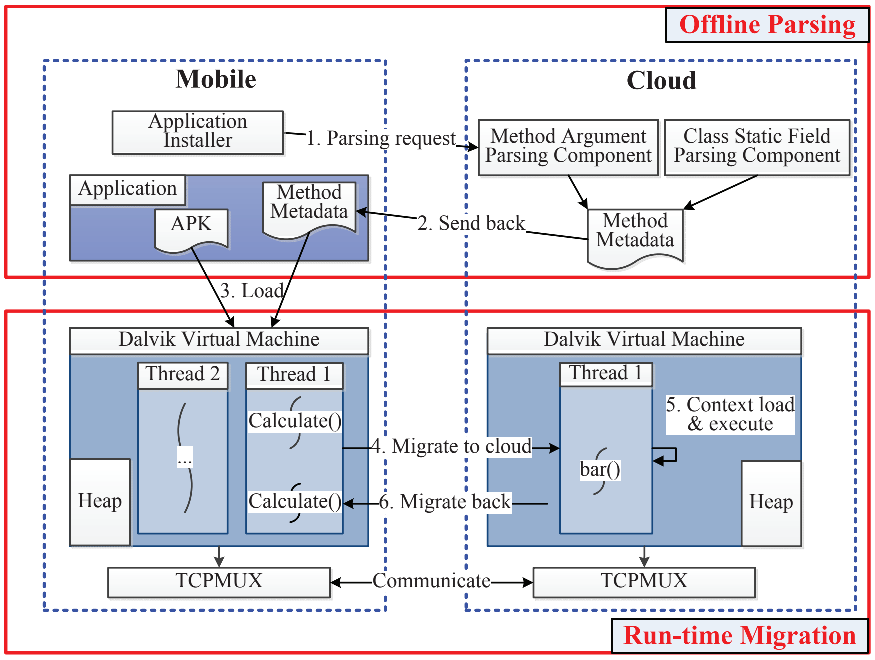


Figure 2.2: The system Architecture of the workload offloading system

2.3.3 Big Picture

In this section, we will describe our system architecture, as shown in Figure 2.2, 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 class static fields. For example, in the method *bar()* of class *Sample* in Figure 2.1(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 2.1(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.

2.4 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

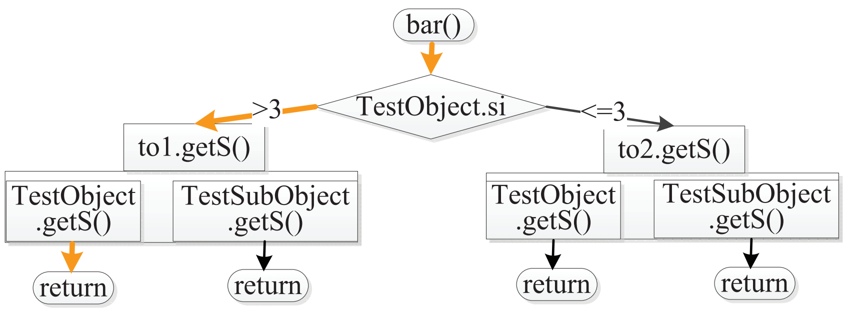


Figure 2.3: Possible code execution paths for *bar()*, orange one is the actual execution path

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.

2.4.1 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 2.1(a) as 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

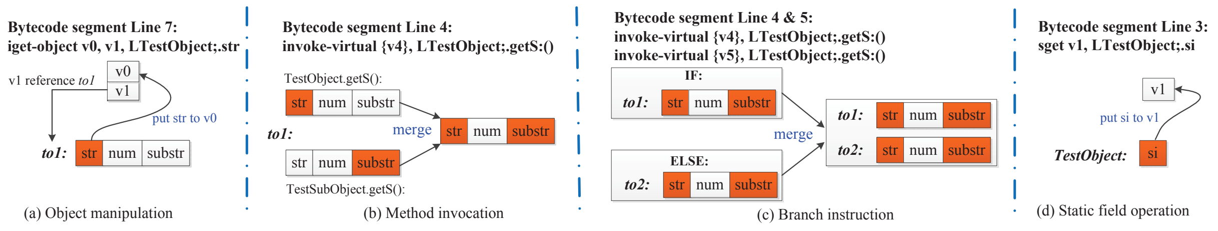


Figure 2.4: Handling the bytecode instructions with respect to Figure 2.1(b)

invoked, the actual code execution path corresponding to the code segment in Figure 2.1(a) is shown in Figure 2.3.

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 [Android Developers] 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.

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 2.4(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.

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 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 2.4(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.

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 2.4(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.

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.

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.

2.4.2 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 Figure 2.4(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 2.1(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()*.

2.4.3 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

Table 2.1: 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

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 Figure 2.1(a), the memory contexts for *bar()* after parsing is shown in Fig 2.5, and the format of the generated metadata from parsing the method *bar()* is described in Table 2.1. 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

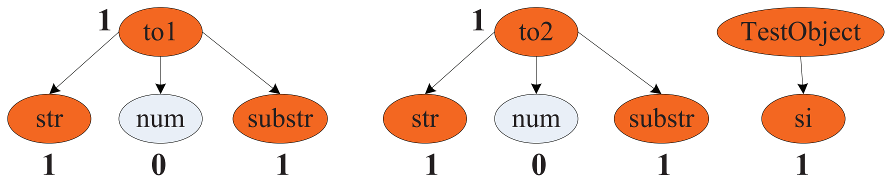


Figure 2.5: The memory context for *bar() after parsing*

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.

2.5 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.

2.5.1 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

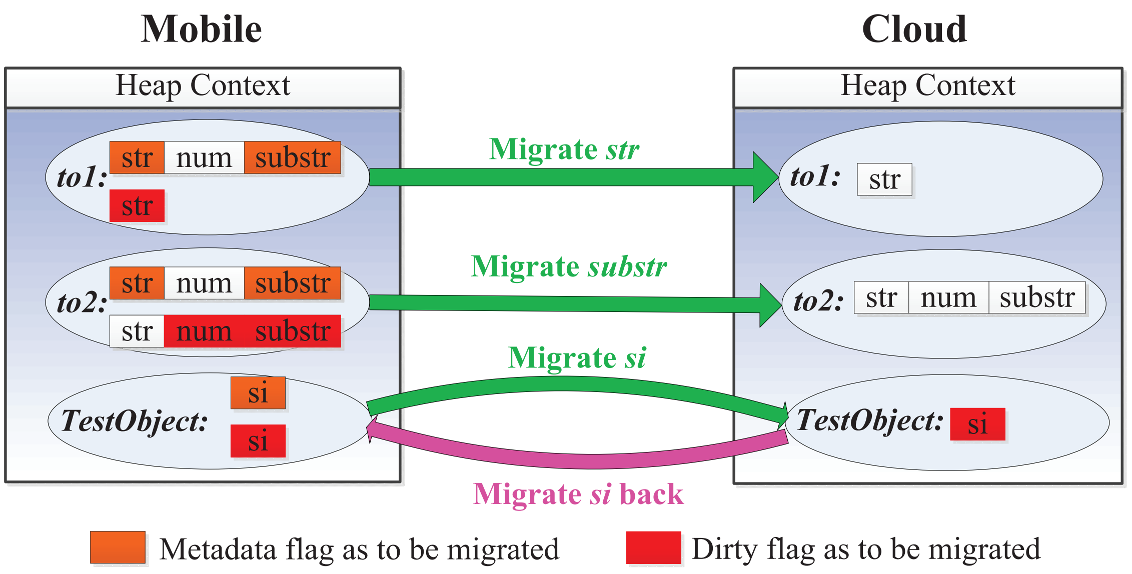


Figure 2.6: VM synchronization during context migration

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.

2.5.2 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 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 Figure 2.6 which corresponds to the invocation of method *bar()* in Figure 2.1(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 2.6, 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 [35] to solve the problem of reference addressing between two endpoints in the executing thread, by assigning an ID to each object during migration.

2.5.3 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.

2.5.4 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 Figure 2.6, 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.

2.6 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.

2.6.1 Experiment setup

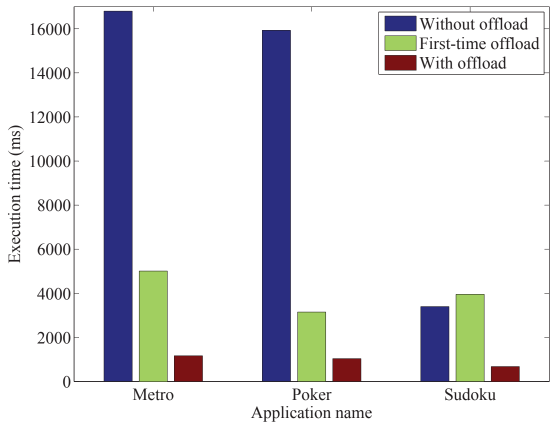
Our experiments are running on Samsung Nexus S smartphones with Android v4.1.2, and a Dell OptiPlex 9010 PC with an Intel i5-3475s@2.9GHz CPU and 8GB RAM as the cloud server. The smartphones are connected to the cloud server via 100Mbps campus WiFi. We use a Monsoon power monitor to gather the real-time 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].

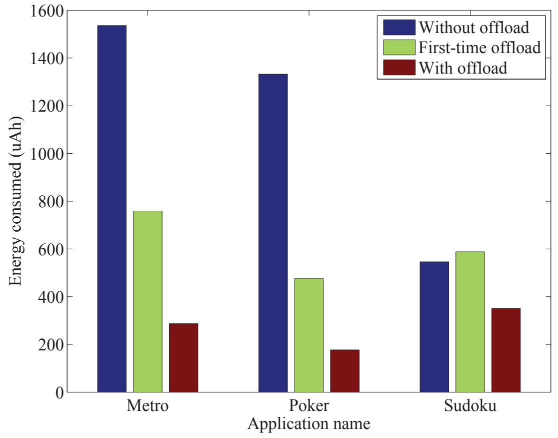
As stated before, our major focus of this chapter is to develop systematic techniques which improve the efficiency of context migration to the remote cloud. Therefore, we do not focus on addressing the problem of *what to offload* and determining the appropriate set of application methods to be offloaded. Instead, in our experiments, we follow the same methodology being used in [35] 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 2.4.1, 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. 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

[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

Figure 2.7: Performance evaluation on execution time and energy consumption

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.

2.6.2 Effectiveness of Workload Offloading

We first compare the method execution time between local and remote executions. From the experimental results shown in Figure 2.7a, 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.

Meanwhile, the local energy consumption is significantly reduced as well by offloading. As shown in Figure 2.7b, 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,

Table 2.2: Amount of data transmission during workload offloading

Application	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

Table 2.3: Offline parsing time and metadata file size

Application	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

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 [35]. The evaluation results are listed in Table 2.2. 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.

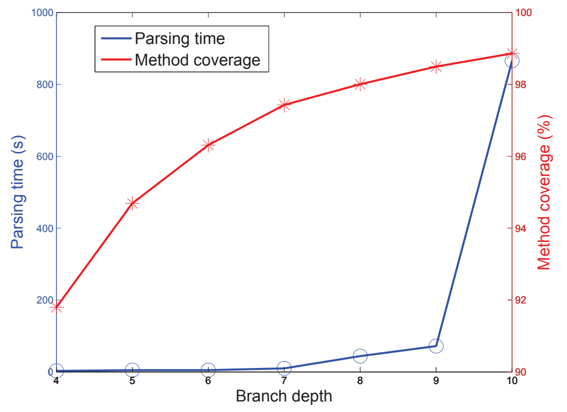


Figure 2.8: Method argument parsing time and method coverage with different parsing thresholds

2.6.3 Parsing complexity

In this section, we evaluate the parsing time and size of metadata of our offline parsing component proposed in Section 2.4. As shown in Table 2.3, 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 2.8, 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.

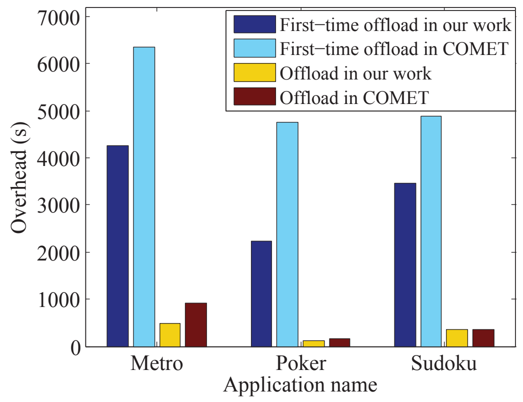


Figure 2.9: Overhead in local mobile when offloading

2.6.4 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 [35]. By seeing the results shown in Fig 2.9, we can tell that the overhead in our system is 30% less than COMET [35] 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 [35] even though we save a significant amount of data transmission.

2.7 Discussion

2.7.1 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 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.

2.7.2 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. In our future work, we will develop a better synchronization mechanism between thread endpoints to better support the current multi-threaded mobile applications.

2.8 Conclusions

In this chapter, 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.

Chapter 3

DeltaVR: Achieving High-Performance VR Dynamics over Mobile Devices through Pixel Reuse

3.1 Introduction

Virtual Reality (VR) stimulates users' immersive senses of the virtual world, and improves user experiences in many interactive scenarios such as gaming [16, 87], automobiles [67], healthcare [24], and education [64]. Ideally, VR should be provided through untethered mobile head-mounted displays (HMDs) that project rendered frames from the connected smartphones, to be usable anytime and anywhere with low cost. However in practice, smartphones have too limited computational capacity and battery lifetime to ensure high rates ($\geqslant$60 FPS), low motion-to-photon latency ($\leqslant$20ms) and wide field of views ($\sim$120° or even 360° panoramic) when rendering high-resolution VR frames [47]. Their VR performance, hence, are much lower than that of their counterparts being tethered to high-performance workstations (e.g., Oculus Rift [26] and HTC Vive [25]).

A viable solution to this challenge is to offload the expensive VR frame rendering to the nearby cloud, and then wirelessly transmit the rendered frame data back to the mobile HMD. Current solutions to mobile workload offloading [19, 22, 50, 35, 34], when being applied to VR applications, fail to provide satisfactory VR performance because their amounts of VR frame data being transmitted may far exceed the capacity of any existing wireless network. For example, the cloud generates more than 2GB of frame data every second for a VR application with 4K resolution and 60 FPS [87], but only a portion of such data can be timely transmitted even through gigabit WiFi network. Video encoding techniques such as H.264 [84] can be used to reduce the size of data transmission, but are too computationally expensive to ensure timely frame decoding at the mobile HMD. Emerging mm-wave wireless could potentially provide the required network bandwidth, but requires line-of-sight connectivity and hence fails to be applied over mobile HMDs, which are mainly used indoor and may move among complicated obstacles.

The fundamental reason of such failure is that the cloud separately renders and transmits every VR frame to the mobile HMD, and the amount of wireless data transmission, hence, is always proportional to the frame rate and resolution of VR applications. The majority of such VR frame data being transmitted, however, could be redundant and wasted in practice. We have experimentally observed that consecutive VR frames are highly correlated, because of *1)* perspective object projection that reduces the impact of user movement on the user view and *2)* image warping that correlates VR frames being projected to different camera locations. Even in highly dynamic VR scenarios such as interactive games, our experiments show that redundancy among consecutive VR frames could exceed 50%, i.e., more than half of pixels in these frames are identical with each other.

Based on this observation, in this paper we present *DeltaVR*, a systematic mobile VR framework that maximizes mobile VR performance by reusing the redundant pixels across consecutive VR frames. Being different from traditional schemes which completely transmit every VR frame to the mobile HMD, DeltaVR only transmits a small portion of VR frames in full, referred to as *reference frames*. For every other frame being produced between reference frames, DeltaVR utilizes the cloud's computational power to explicitly decide its overlap with the last reference frame, and only transmits its distinct portion (as a *delta image*) to the mobile HMD afterwards. By doing this, DeltaVR completely eliminates the redundancy in the VR frame data being transmitted, and has the following two unique features that ensure high-performance mobile VR with highly dynamic application contents and user behaviors.

First, DeltaVR is widely applicable to highly dynamic VR applications, especially interactive VR games where the user character constantly moves and interacts with foreground objects in the virtual world. DeltaVR, hence, fundamentally outperforms existing mobile VR solutions, which are limited to reusing pixels of the panoramic background image only when the user character remains stationary in the virtual world [23, 16, 52]. In such dynamic VR scenarios with fluctuating user views, to ensure correct pixel mapping from a reference frame when applying the delta image, we first transform the pixels of the reference frame to the target user view via image warping, and then use the delta image to patch the visual artifacts during the process of image warping. In this way, DeltaVR significantly extends the scope of pixel reuse without impairing the VR image quality.

Second, DeltaVR maximizes the performance of highly dynamic VR applications without predicting the user behavior in the virtual world or prefetching any VR frame data based on such prediction. Instead, the mobile HMD in DeltaVR generates every VR frame based on the corresponding delta image at run-time. As a result, being different from traditional schemes whose performance could be easily impaired by sporadic events or user movements in VR applications [54, 23, 53], the performance of mobile VR applications supported by DeltaVR is only determined by their view resolution and wireless link condition that decides the VR frame rate.

The major challenge of designing DeltaVR, on the other hand, lies in how to minimize the communication and computation latency for the mobile HMD to receive and process the delta images. First, a reference frame in VR applications is usually panoramic to provide seamless 360° user views, and is much larger than any regular frame being displayed on the smartphone screen. In this case, to reduce the amount of wireless data transmission, we constrain the scope of calculating delta images within the specific user field of view (FOV) in the virtual world. Second, to minimize the computational overhead of image warping over panoramic reference frames, we also develop new techniques to partially warp the portion of a reference frame within the user FOV.

We have implemented DeltaVR over the Android OS and Unity VR application engine[1] as a mobile middleware between VR applications and OS drivers, so as to ensure its

[1]The Unity engine (https://unity3d.com/) is the most popular tool for commercial VR game creation.

generality over different VR applications with heterogeneous dynamics and computation demands. More specifically, DeltaVR is implemented in native language and we utilize the unified OpenGL APIs for graphics operations such as VR frame rendering, so as to tackle the heterogeneity of shading languages and scripting APIs used by different VR applications. The implementation consists of ∼4,000 Lines of Codes (LoC) in total, and our experimental results over real-world VR applications show that DeltaVR can maximize the VR performance over highly dynamic VR scenarios with complicated scenes and intensive user movement, while reducing more than 95% of the VR frame data being wirelessly transmitted.

3.2 Motivation & Preliminaries

Our design of DeltaVR is motivated by the unique characteristics of frame rendering in VR applications. First, the impact of slow user movement on the user view could be reduced by *perspective projection* in VR applications. Such reduction results in very high redundancy across consecutive VR frames, which minimizes the sizes of delta images and ensures timely transmission of these images to the mobile HMD. Second, for fast user movement in the virtual world, a significant portion of redundant pixels can still be retained by *image warping*, which reprojects a rendered VR frame based on user movement during the rendering process. Exploiting such redundancy in VR image warping could greatly expand the scope to which DeltaVR is applied, by minimizing the amount of reference VR frames being produced and sent.

3.2.1 3D Perspective Projection

As shown in Figure 3.1, VR applications construct the virtual world as a 3D space, where game objects are modeled and placed at certain coordinates. In this 3D world, the user character is represented by a 2D camera, and the application view being presented to the user is rendered by projecting each 3D object to the camera surface. Specifically, most of today's VR applications adopt perspective projection [66, 27], which emulates how human eyes see the real world. Such projection forms the 3D world as a truncated pyramid frustum,

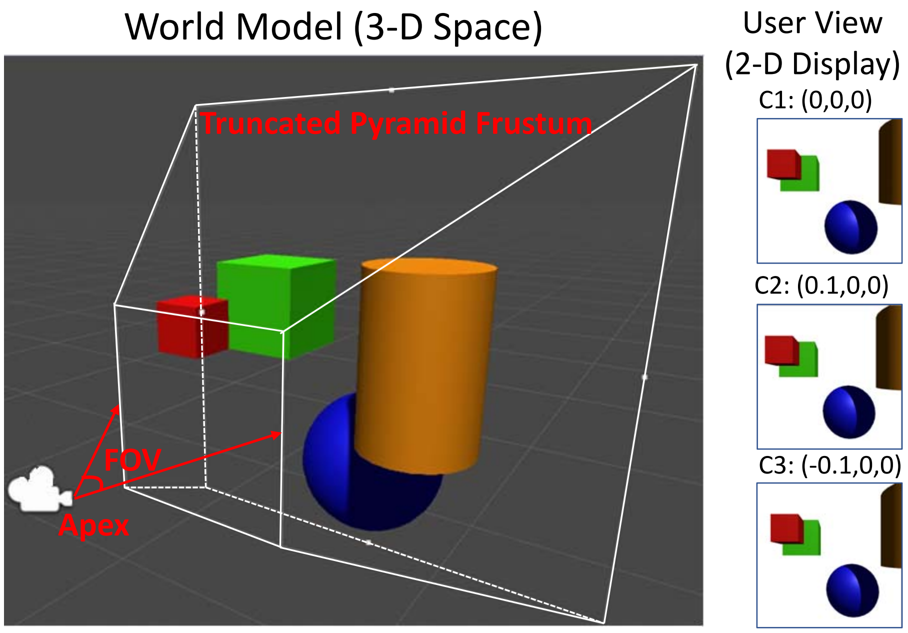


Figure 3.1: The virtual world in VR applications

with the camera sitting at the apex point and its range being defined as the camera's FOV. Any object within this frustum is projected to and visible in the user view.

The most significant characteristic of perspective projection is that distant objects in the 3D world appear smaller than objects close-by, and the impact of user movement on the 2D user view will hence be reduced after object projection. In many cases, such impact could be as small as few pixels and results in high volumes of redundant pixels across consecutive VR frames. For example, when the VR user changes his/her head orientation, a neck model is adopted to measure the camera location change, which usually ranges in [-0.1, 0.1] in virtual units (~10cm in reality) and results in negligible user view change as shown in Figure 3.1 (from camera location C1 to C2). Slow user movement at 1 m/s, on ther other hand, could result in more than 90% redundant pixels across consecutive frames [65].

3.2.2 VR Image Warping

In highly dynamic VR applications, the user character may move to a different location in the mean time when a new VR frame is being rendered. To ensure image quality and avoid motion sickness, image warping is adopted by VR applications to reproject a rendered frame to the new camera view before displaying. For example, the most commonly used

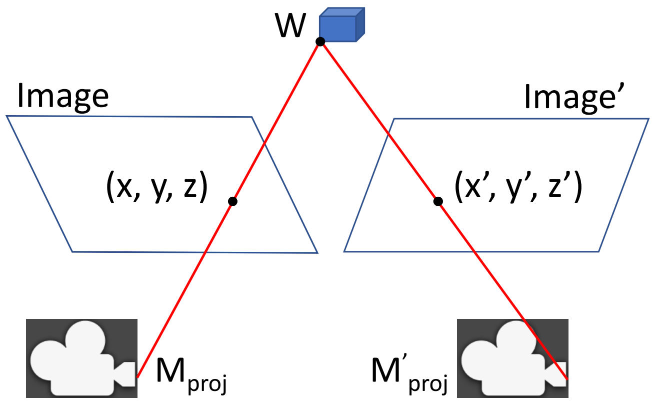


Figure 3.2: VR image warping

(a) Viking Village **(b)** Lite **(c)** Sci-Fi

Figure 3.3: Screenshots of VR games

technique, *Image-based Rendering (IBR)* [63, 69], is illustrated in Figure 3.2. For any pixel (x, y) on the 2D user view plane, its coordinate in the 3D virtual world can be computed as $W = M_{proj}^{-1} \cdot (x, y, z)$, where z is the depth value of (x, y) and M_{proj} indicates the current camera projection. Then, when the camera projection changes to $M_{proj}^{'}$, the new user view can be produced by reprojecting W onto the 2D plane as $(x', y', z') = M_{proj}^{'} \cdot W$ for every pixel, without re-rendering these pixels at new locations.

Such reprojection naturally correlates VR frames being projected to different camera locations. To quantitatively study such correlation, we randomly pick 10 reference frames from three open-sourced VR games (Viking Village [vik], Lite [lit] and Sci-Fi [sci]), and utilize IBR to warp these frames to the target camera views from different distances away. As shown in Figure 3.3, these games represent different VR scenarios with heterogeneous scene complexity and character dynamics, and Figure 3.4 demonstrates that more than 50% of pixels can be retained in the VR frames after image warping, even if the warping distance

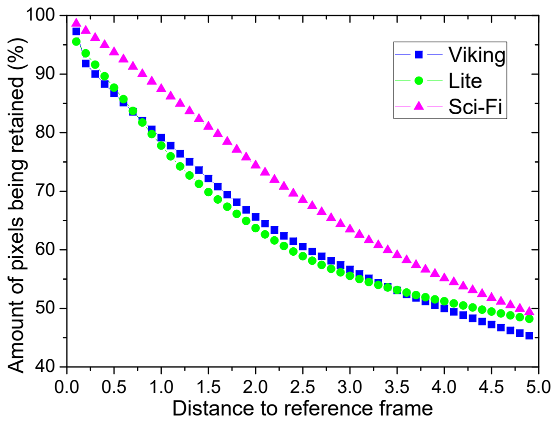


Figure 3.4: Frame correlation after image warping

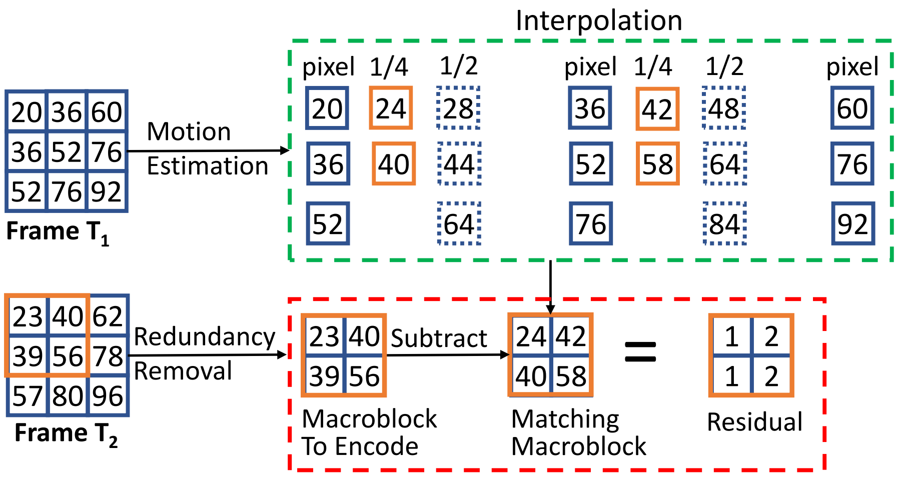


Figure 3.5: Illustration between motion estimation and image warping

increases to 5.0 (∼5m in reality). In other words, VR image warping retains a significant portion of redundant frame pixels over long distance. Such redundancy is exploited in DeltaVR to minimize the amount of reference VR frames being transmitted, by generating multiple delta images at different camera locations over time from the same reference frame.

3.2.3 Failure of Traditional Video Encoding

One straightforward solution to eliminating such VR pixel redundancy is to adopt the existing video encoding techniques for VR frame compression. Traditional video encoding techniques, unfortunately, fail when being directly applied to VR frames, because they are generally agnostic about the object projection structure of VR frames and are hence incapable of adapting to the heterogeneous VR dynamics. For example, as the most commonly used technique, H.264 [84] compresses video frames based on their temporal and spatial locality. More specifically, H.264 divides a frame into macro blocks (MBs), which serve as the basic unit of encoding. For each MB, H.264 utilizes the previous encoded frames as reference and estimates the user motion to find the region that most closely matches the current MB. Afterwards, the difference to the matching MB would be compensated by their residual, which is then quantified for different compression ratios and entropy encoded with context-adaptive variable-length coding.

Since the user motion may not always displace the corresponding pixels at integer-pixel level, computation intensive interpolation [21] has to be performed to search the matching MB at fractional-pixel level. However, due to the finite computational capacity, such pixel interpolation during motion estimation in H.264 is limited to quarter-pixel level at the finest granularity. Such limitation correlates pixels between VR frames inaccurately and incurs large overhead in computation and communication.

We use the video encoding of two frames in Figure 3.5 as an illustration. When a VR frame T_1 is utilized as the reference to encode the VR frame T_2, the motion estimation would find the best matching MB by interpolating the frame T_1 for quarter-pixel values, which takes about 8 arithmetic operations for each pixel [21]. The residual MB after subtraction, nonetheless, fails to eliminate the frame redundancy and is still in high entropy, which takes about 7 bits to be encoded in H.264. In contrast, the image warping could accurately reproject the pixels of frame T_1 to the new camera position and completely removes the pixel redundancy between frames. Consequently, the resulting MB would consist of pixel values of zero and can be processed more effectively by existing video encoding techniques because the MB requires no interpolation during motion estimation.

3.3 Overview

Figure 3.6 illustrates how DeltaVR works: it exploits and removes the redundancy across consecutive VR frames whenever possible, so as to only transmit the delta images with minimium sizes to the mobile HMD for VR display. At the cloud, DeltaVR periodically

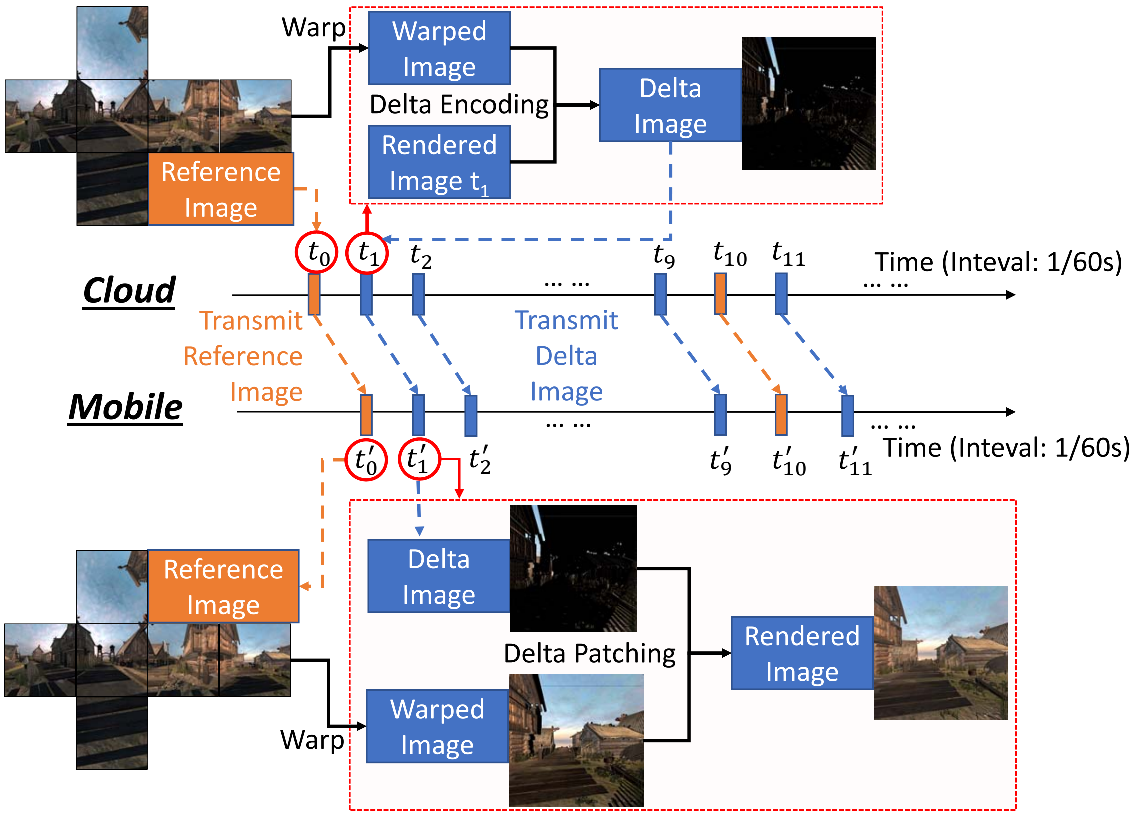


Figure 3.6: Overall design of DeltaVR

renders a panoramic image that captures all the possible user orientations at the current camera position in the virtual world. Such a panoramic image is then used as the reference frame to synthesize delta images for other VR frames in the future. Every time when a new VR frame is needed[2] at time t_1, DeltaVR first renders this frame in full at the cloud, and then warps the most recent reference frame from its original camera view at t_0 to the current user view at t_1. As a result, the delta image of this frame at t_1 is synthesized via delta encoding as the difference between the originally rendered frame and the warped image from the reference frame.

How to minimize the amount of VR frame data being transmitted? As shown in Section 3.2.2, a large amount of redundant pixels can be retained across consecutive VR frames or even after image warping over long distance. Hence, delta images for multiple VR frames can be synthesized from the same reference frame, and the size of each delta image is always smaller than the corresponding full VR frame. In practice, the sizes of delta images will grow when the user character keeps moving and results in longer warping distance. In order to ensure timely transmission of each delta image within T, DeltaVR further reduces the average size of delta images to <25 KB without impairing the VR image quality, through image compression and clipping (see Section 3.4.1).

How to maximize mobile VR performance? At the mobile HMD, DeltaVR warps the received reference frame in the same way to the user view at t_1. When the corresponding delta image is received, it applies the delta image to patch the visual artifacts being produced by image warping, so as to restore the full VR frame for display without any image quality loss. In this way, DeltaVR avoids the expensive rendering of any VR frame pixel at the mobile HMD, and hence guarantees the VR frame rate regardless of the scene complexity or level of graphics quality in VR applications. The major factor that may affect the mobile VR performance, then, is the large size and high resolution of panoramic reference frames, which incur higher computation overhead for image warping at the mobile HMD. To address this problem, DeltaVR adaptively clips each panoramic reference frame and only warps its portion within the current user view to the mobile HMD (see Section 3.4.2).

[2]The time interval (T) between two consecutive VR frames is determined by the frame rate, which should be at least 60 FPS for satisfiable VR performance.

Besides, in practical VR scenarios, when the user character keeps moving during the mean time when the delta image is being transmitted, the VR frame being displayed may not precisely capture the user movement but falls behind. The consequent lag in VR display, however, is minimum because of the prompt transmission of small-sized delta images, and could be easily addressed by re-warping the patched VR frame to the current user view.

3.4 Delta Image Synthesis

Efficient synthesis of delta images is the core of DeltaVR to satisfy the performance requirement of today's VR applications. First, DeltaVR synthesizes delta images with the minimum size at the cloud, so that each delta image could be timely transmitted to the mobile HMD. Second, DeltaVR also minimizes the computation overhead of applying delta images at the mobile HMD, to ensure that each VR frame can be timely restored and displayed to the user.

3.4.1 Minimizing the Delta Image Size

DeltaVR synthesizes a delta image through per-pixel subtraction between the full VR frame and the warped image from the corresponding reference frame. Specifically, for VR frames with 8-bit pixel channels[3], the pixel value in each channel of the delta image is computed as

$$Delta = \frac{Full - Warped}{2} + 127, \tag{3.1}$$

which maps the positive and negative differences between the full VR frame and the warped image to lighter and darker colors, respectively. Similarly, when restoring the full VR frame, the mobile HMD patches the delta image to the warped image by inversing the subtraction as

$$View = \min[2 \cdot (Delta - 127) + Warped, 255]. \tag{3.2}$$

Based on such encoding, DeltaVR further reduces the size of delta image from the following two aspects. First, the cloud compresses each delta image before sending it, and

[3]The pixel value in a 8-bit channel ranges from 0 to 255.

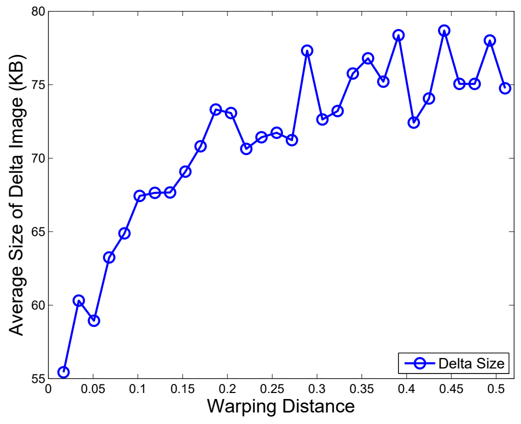


Figure 3.7: Average size of delta images after compression

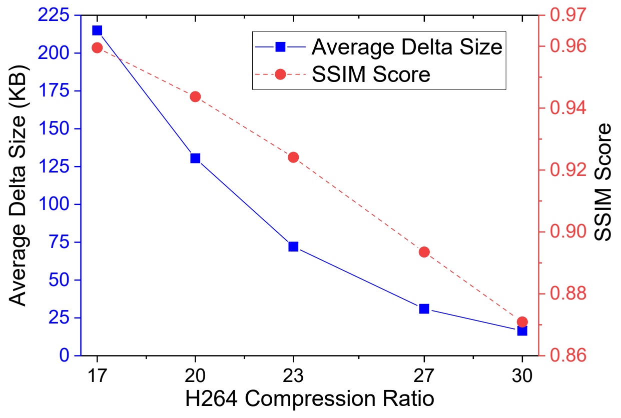


Figure 3.8: Balancing between delta size and image quality

uses the decompressed version of compressed reference frames at the cloud to ensure the consistency of delta image synthesis with the mobile HMD. Second, the cloud clips each delta image according to the current user camera orientation and FOV, In this way, it avoids transmitting any VR frame data outside of the current user view, which is unlikely to be noticeably changed during the short time period of transmitting a delta image. Our experimental studies show that these techniques could reduce the size of a delta image to $<$ 25 KB without any VR image quality loss. Such reduction, on the other hand, also allows a reference frame to be used for synthesizing more delta images and further minimizes the total amount of VR frame data being transmitted.

Delta Image Compression

The most straightforward approach to reduce the size of a delta image is to compress the image at the cloud before transmitting it to the mobile HMD. Since the size of a delta image is much smaller than the full VR frame, each delta image, after being processed by existing lossy compression techniques such as H.264 [42], could be efficiently decompressed by the mobile HMD before the next delta image arrives. As shown in Figure 3.7, the average size of delta images with H.264 compression continuously increases along with the warping distance, which reduces the amount of redundant pixels in VR frames when it increases. Even when the warping distance is very long (∼0.5), such average size could be lower than 80 KB with the default compression ratio (23)[4].

However, applying such a lossy compression technique over delta images in DeltaVR is challenging, because it may result in discrepancy in delta image synthesis between the cloud and the mobile HMD, further impairing the VR image quality. More specifically, the cloud synthesizes a delta image by warping from a uncompressed reference frame, but has to send such a panoramic reference frame to the mobile HMD after compression. The warped image from the decompressed reference frame at the mobile HMD, hence, will have more visual artifacts due to lossy data compression and affect the correctness of delta patching.

To address this challenge, DeltaVR retains a decompressed version of each compressed reference frame, and uses this version for image warping at the cloud to ensure consistency of delta image synthesis. The correctness of delta patching at the mobile HMD, then, could only be impacted by compression over the delta images themselves. In practice, such impact can be controlled by adopting different H264 compression ratios that balance between the VR image quality and delta image sizes. To evaluate such balance, we conducted preliminary experimental studies by using the structural similarity (SSIM) metric [83] over the Viking Village VR game [vik]. According to [23], SSIM is designed to model the human eye's perception to 3D images, and a SSIM score higher than 0.9 indicates good quality of VR images. Our experiment results in Figure 3.8 show that any H264 compression ratio lower than 27 would result in a satisfiable level of VR image quality, and could further reduce the average size of delta images down to 25 KB.

Delta Image Clipping

The size of delta image could be further reduced by exploiting the limited FOV of today's mobile HMDs, which is usually smaller than 120° [fov]. As a result, instead of synthesizing and transmitting a delta image over the 360° panoramic view, DeltaVR transmits to the mobile HMD with a clipped delta image corresponding to the current user camera orientation

[4]H.264 allows different compression ratios by adjusting its Constant Rate Factor (CRF), which decides the amount of data bits being used for each image frame.

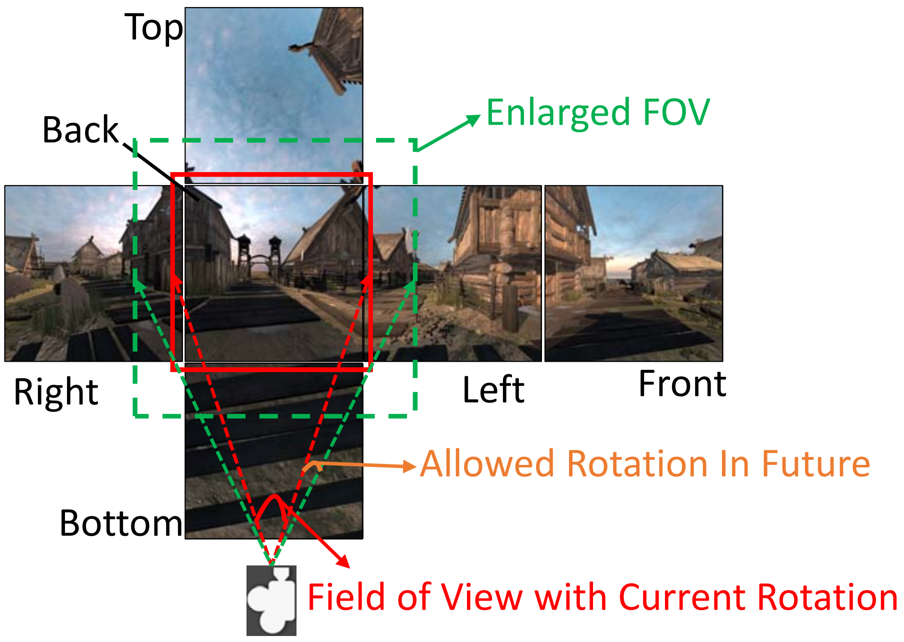

Top
Back
Enlarged FOV
Right
Left
Front
Allowed Rotation In Future
Bottom
Field of View with Current Rotation

Figure 3.9: Delta image clipping

and FOV, which are reported from the mobile HMD to the cloud every time when a new VR frame is needed.

The major challenge of such delta image clipping, however, is that the user view may change during the process of delta image synthesis due to user head rotation, and such change cannot be known by the cloud in advance. Our solution to this challenge, as shown in Figure 3.9, is to further enlarge the FOV of image clipping by $X°$ in both sides, to cover the possible change of user view. In practice, since each delta image is promptly transmitted to the mobile HMD within a very short amount of time, the possible change of user view during this short time period is very limited. For example, even with the most vigorous user head rotation where the angular velocity reaches to 780° per sec [37], the value of X is merely 17.5 for a 22ms latency of delta image transmission.

As shown in Figure 3.10, after H.264 compression, such clipping further reduces the size of delta images by up to 65%, when being applied to the three open-sourced VR games that we described in Section 3.2.2. In particular, such size could be effectively controlled within 25 KB when the user FOV is smaller than 150°, which could be considered as the optimal FOV that well balances between VR frame rate and user experience in practice.

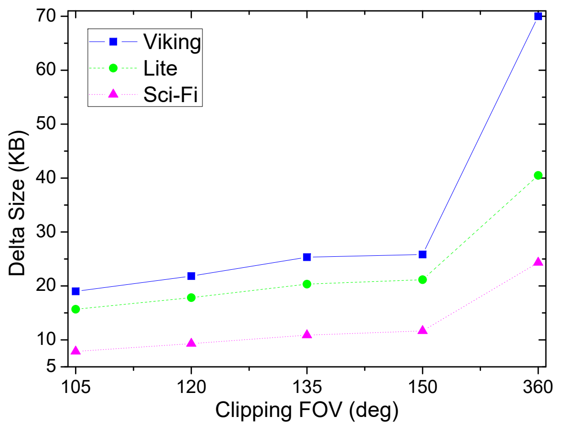


Figure 3.10: Delta size with different clipping FOV. H.264 with CRF=23 is being used.

3.4.2 Minimizing the Computation Overhead

After having received a reference frame and the successive delta images, the performance of DeltaVR depends on the mobile HMD's computational capability of image warping and delta patching. While delta patching can be done via texture mapping with low complexity, warping over reference frames with 360° panoramic view and ultra-high resolution could be too expensive to be affordable at the mobile HMD[5]. According to our experimental studies, it takes up to 3.35 ms for a LG G5 smartphone to warp a reference frame with 1024x1024 resolution over the warping distance of 0.5, hence restricting the maximum VR frame rate to be lower than 30 FPS.

Intuitively, such computation overhead could be reduced by the same approach of image clipping in Section 3.4.1, which warps only the portion of a panoramic reference frame that is visible within the user view. The major challenge, however, is how to decide such portion without impairing VR user experience or image quality. Since the user character in the virtual world may move during the warping process, if we clip a reference frame only based on the current FOV and camera orientation at the mobile HMD, certain visible regions in the new target view may be missed. Instead, DeltaVR exploits the property of perspective projection in VR applications, and determines the right FOV of clipping a reference frame according to the real-time user movement in the virtual world.

[5]According to [48], the computational complexity of image warping is proportional to the size and resolution of the reference image.

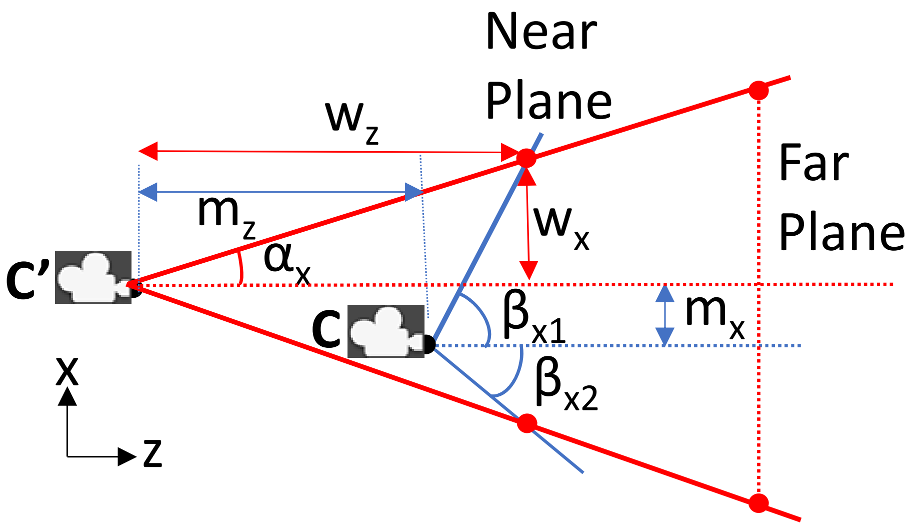


Figure 3.11: FOV for reference image clipping

According to perspective projection, the 2D user view plane in VR applications is rendered with the VR objects between the near plane and the far plane in the truncated pyramid frustum. As shown in Figure 3.11, when the user character moves from the current camera position **C** to a new position **C′** during the warping process (only showing the X-Z plane for simplicity), the FOV for reference image clipping at **C′** should make sure that the frustum at **C** (defined as $\beta_{x1} + \beta_{x2}$) is fully covered. According to the Pythagorean theorem [61], the values of β_{x1} and β_{x2} can be determined as

$$\begin{cases} \beta_{x1} = \arctan \frac{w_x + m_x}{w_z - m_z}, \\ \beta_{x2} = \arctan \frac{w_x - m_x}{w_z - m_z}, \end{cases}$$

where w_z is the distance of near plane, $w_x = w_z * \tan \alpha_x$, and α_x is half of the FOV at **C′**. Similarly, the FOV for reference image clipping in the Y-Z plane can be computed in the same way.

3.5 Implementation

We implement DeltaVR over Google VR Unity SDK v1.20 and Unity VR application engine v5.5.1, with minimum modification on either the Google VR SDK itself or the VR application binaries. It consists approximately 3100 lines of C++ code as a plugin to the Unity engine, and 850 lines of C# code as a Unity engine script. We use x264 [x26] as the encoder and decoder of delta images.

3.5.1 Cloud Operations

DeltaVR runs a clone copy of each VR application at the cloud, and renders VR frames according to the user inputs such as the controller operations received from the mobile HMD as system events. To retrieve the rendered full VR frames from the application binary, we exploit the hook of the application engine and attach a post-processing script to the specialized VR camera. This script transforms the depth buffer into a greyscale image, and then reads the raw pixels of the color and depth images into the main memory.

On the other hand, in order to render the panoramic reference frames at the cloud, we create a specialized camera in the VR application binary, which utilizes the VR application engine's API to render the scene as a cubemap. Specifically, the camera renders the scene onto the sides of a cube with six square textures, which represent the view along the directions of the world axes (up, down, left, right, forward and back). Each face of the cubemap has a FOV of 90° and a resolution of 1024x1024 so as to capture a 4K panoramic view of the scene.

3.5.2 Mobile OS Integration

The major challenge of DeltaVR implementation at the mobile HMD is how to efficiently support different VR applications in a generic manner. First, VR applications are heterogeneous in their shading languages and scripting APIs being used. For example, the Unity engine uses either JavaScript or C# as the script language, but the Unreal engine[6] only supports C++. Supporting pixel reuse within the VR application binary, hence, leads to repetitive efforts of re-programming. Second, operations of pixel reuse, if being done in the user space, would be less effective due to frequent interaction with the system hardware.

To address these challenges and retain generality, we integrate DeltaVR into the OS kernel of the mobile HMD, and implement it as a middleware between VR applications and

[6]https://www.unrealengine.com/

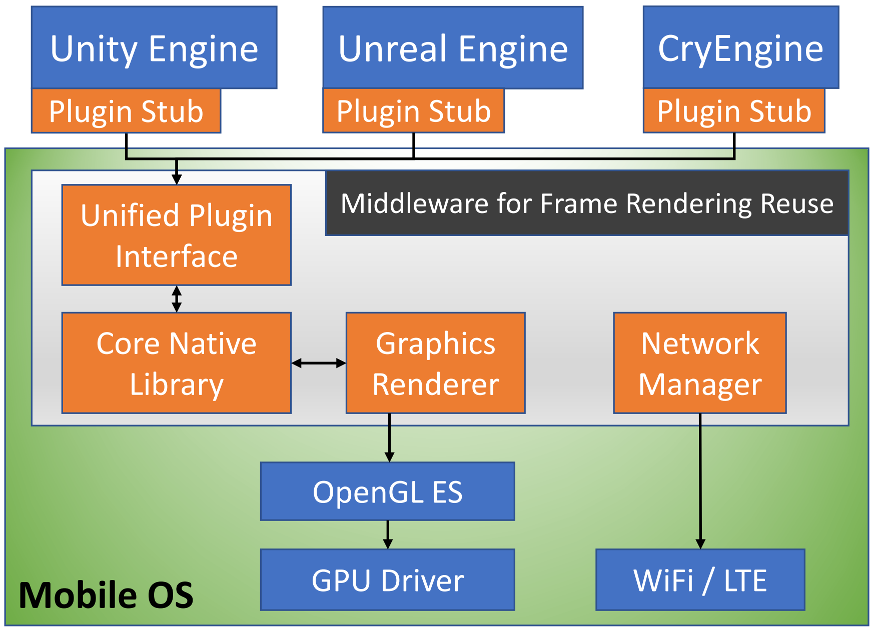


Figure 3.12: DeltaVR as a mobile middleware

OS drivers. As shown in Figure 3.12, the core of DeltaVR is implemented as an OS library in native language to regulate the main DeltaVR functionality. The core library then interacts with the graphics renderer, which manages frame buffers and invokes APIs directly from OpenGL ES for delta patching and image warping. Since the OpenGL provides unified APIs for 3D graphics rendering, the pixels in VR frames are generically reused without involving the engine-specific shading languages such as the Microsoft's HLSL [81] and Nvidia's Cg [62].

On the other hand, the core library needs to interact with VR app binaries to retrieve the necessary metadata for pixel reuse, such as the current camera position, orientation and FOV. An intuitive solution is to invoke engine-specific APIs directly from the core library, but lacks generality.

Instead, we introduce a middle layer with a suite of unified plugin APIs for data exchange as shown in Figure 3.13. In particular, a plugin stub is implemented with engine-specific scripts to fulfill behaviors of the predefined APIs. Such stub is dynamically linked with the core library during development, so that any invocation to the plugin API will be directed to the plugin stub at runtime. For example in Unity, to warp the reference frame to the target view at runtime, the graphics renderer in the core library needs to find out the current

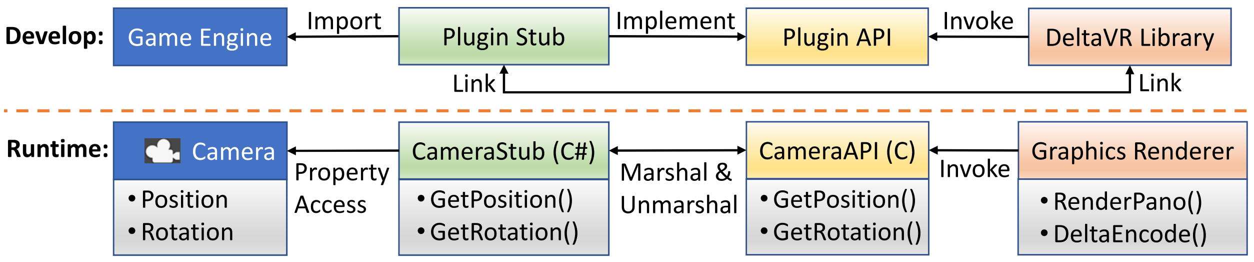


Figure 3.13: Unified interaction with game engine

camera position and hence will invoke the *GetPosition()* function in the plugin *CameraAPI*, which is written in native C. This function marshals the request to the managed format in C#[7] and triggers the engine-specific script *CameraStub* to access the position property of the camera object. Afterwards, the position values of the engine camera is marshaled to the native format and returned to be processed by the graphics renderer.

3.5.3 Parallel and Pipeline Processing

DeltaVR is also implemented to be affordable at low-end mobile devices with severely constrained computation capabilities, especially the older generations of smartphones being used as the mobile HMD. In order to efficiently utilize the limited system resources and reduce the response latency on these challenging systems, we divide the DeltaVR operations into individual tasks and execute them in a pipeline manner for the two stereo eyes in each frame. As shown in Figure 3.14, when the system is working to render and warp for the right eye of frame 0 (denoted as R_0), it is simultaneously encoding the delta image for the left eye of frame 0 (L_0). To avoid pipeline stalls or resource idleness due to the heterogeneous computational complexity in different stages, we maintain a request queue for each stage, which can then proceed to the next task immediately without waiting. In addition, we also share the VR frame memory and allow the memory handle to be passed between stages, so as to avoid copying the bulky VR frame data itself.

With the pipeline, the mobile VR performance is constrained by the most computationally expensive stage in the pipeline, whose processing time is further reduced in DeltaVR by exploiting the system parallelism. In particular, when the limited GPU resources on

[7]http://msdn.microsoft.com/en-us/library/ms235282.aspx

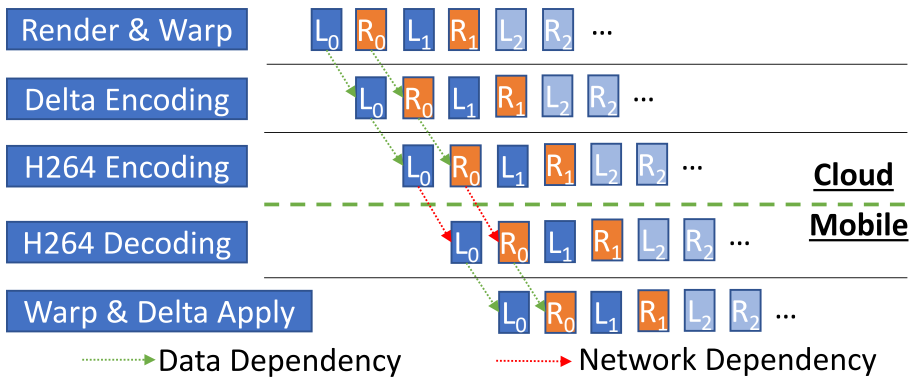


Figure 3.14: Pipeline processing of DeltaVR

low-end mobile HMDs are fully used by image warping and hence incapable of decoding the compressed delta images timely, DeltaVR splits a delta image into multiple segments and dedicates specialized CPU threads for faster software decoding.

3.6 Making VR Apps with DeltaVR

The generic design and implementation of DeltaVR significantly reduce the burden of VR application development, with the Unity engine as the target VR software platform. Typically, as shown in Figure 3.15, the Unity engine converts the application-specific 3D objects and scripts of user interaction into native codes that are further compiled as executable binaries, so as to render the VR scenes at run-time. Such procedure enables the application developer to easily extend the application's functionality from a basic prototype by simply linking the new native libraries into the existing program binaries.

Our work exports the components implemented in Section 3.5.2 as easy-to-use modules, based on which VR applications can be built for both the cloud and the mobile HMD. As shown in Figure 3.15, the developers simply need to import the modules provided by DeltaVR by copying the libraries to the application folder and create special prefab[8] instances in the Unity engine, and these prefabs will then be dynamically linked into the final executable at compile time. Specifically, besides the core library, the modules of image warping and delta encoding/decoding should also be included into the graphics renderer at both the cloud and

[8]A game object acts as a template with predefined scripts and properties.

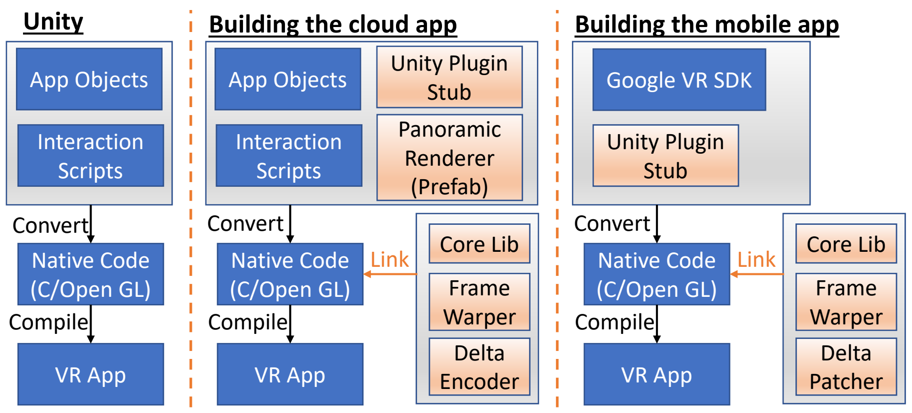


Figure 3.15: Making VR apps with DeltaVR

the mobile HMD, and a prefab of panoramic renderer should be created at the cloud to render panoramic reference frames.

3.7 Evaluation

In this section, we evaluate the performance of DeltaVR, which is measured by the VR frame rate, image quality and motion-to-photon latency. Our experiment results show that DeltaVR can maximize the VR performance over highly dynamic VR scnearios with complicated scenes and intensive user movement, and also demonstrate the effectiveness of DeltaVR in resource-constrained scenarios with limited wireless network bandwidth and device energy budget.

3.7.1 Experiment Setup

In our experiments, we use a LG G5 smartphone with Android v6.0.1 as the mobile HMD, and a Dell OptiPlex 9010 Desktop PC with an Intel i5-3475s@2.9GHz CPU, Radeon HD 7470 GPU and 8GB RAM as the cloud server. We use a Google cardboard as the experimental VR headset with a FOV of 90°. The mobile HMD is connected to the cloud server via campus WiFi, which has an average throughput of 40 Mbps and transmission latency of 3.5 ms. A Monsoon power monitor[9] is used to measure the energy consumption of the mobile HMD, and each experiment is conducted multiple times for statistical convergence.

[9]https://www.msoon.com/LabEquipment/PowerMonitor/

Table 3.1: Statistics of VR scene complexity

Game	Draw Calls	Triangles (K)	Vertices (K)
Viking	400	2,400	1,600
Lite	212	65.7	52.4
Sci-Fi	227	32.7	36.7

Our experiments are conducted over the three open-sourced VR games listed in Section 3.2.2. As shown in Table 3.1, they present different levels of VR scene complexity. Unless explicitly stated, each VR scene contains 4 animated foreground objects moving at 1m/s towards a random direction, and the user character constantly moves at the same speed in the virtual world. By default, DeltaVR transmits a new reference image to the mobile HMD every 60 VR frames, resulting in a maximum warping distance of 1 in the virtual world. Each panoramic delta image, before being transmitted, is clipped with a FOV of 135°, which allows a 22.5° head rotation with Google cardboard and tolerates 28 ms delay for transmission and decoding. X264 with default CRF=23 is being used for delta encoding and decoding.

We compare DeltaVR with three existing VR schemes:

- **Local:** VR applications are solely running on the mobile HMD.
- **Thin-client**: Every VR frame is rendered by the cloud and transmitted in full to the mobile HMD [wow].
- **Furion**: A VR frame is collaboratively rendered at the cloud and mobile HMD. Panoramic VR backgrounds are rendered at the cloud and pre-fetched by the mobile HMD for all possible directions of user movement. Foreground VR objects are all rendered at the mobile HMD itself [52].

3.7.2 Improvement of VR Performance

Our experiment results show that, by avoiding expensive VR frame rendering at the mobile HMD, DeltaVR always achieves the required 60 FPS with different levels of VR resolution and scene complexity, while providing high image quality with SSIM > 0.92. It also

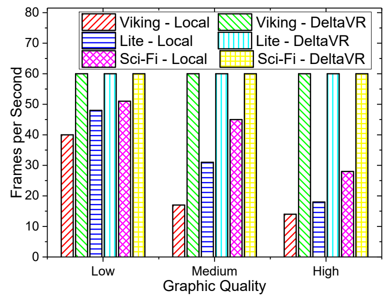


Figure 3.16: Frame rate with different settings of graphic quality

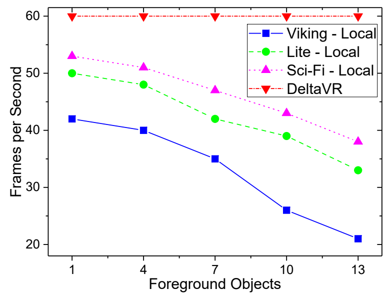


Figure 3.17: Frame rate with different levels of VR scene complexity

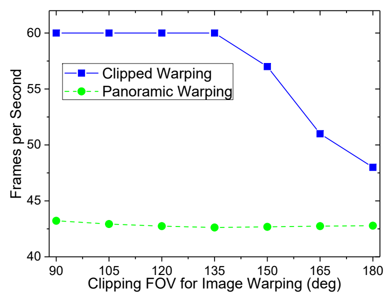


Figure 3.18: Frame rate with different FOV for image warping

minimizes the motion-to-photon latency within 16ms to ensure smooth VR experience and avoids any possible motion sickness.

Frame Rate

As shown in Figure 3.16, the frame rate provided by DeltaVR is constantly 60 FPS in all settings of graphic quality, and greatly outperforms local VR frame rendering whose performance significantly drops to < 15 FPS under high resolution. Similarly, Figure 3.17 shows that the frame rate in DeltaVR remains constant even when the VR scene becomes highly complicated with 13 foreground objects. In contrast, the local VR frame rendering experiences more than 50% performance degradation by rendering the additional foreground objects at the mobile HMD. Note that, the maximum FPS that DeltaVR can achieve in our experiment is limited by the screen refreshing rate at the mobile HMD that is being capped at 60Hz, and could hence be further improved on future mobile devices which supports higher screen refreshing rates (e.g., 90Hz).

One reason for such improved VR performance, as described in Section 3.4.2, is the clipping process over panoramic reference frames that reduces the computation overhead of image warping at the mobile HMD. As shown in Figure 3.18, compared with panoramic image warping that reduces the VR frame rate down to ~ 42 FPS, DeltaVR improves the frame rate by more than 40% as long as the clipping FOV does not exceed $135°$, hence allowing a maximum warping distance of 0.88 without any VR performance degradation.

Table 3.2: VR image quality (SSIM)

Rendering Scheme	Viking	Lite	Sci-Fi
Local Frame Rendering	0.8133	0.8766	0.8832
DeltaVR w/ Stationary User	0.9569	0.9599	0.9681
DeltaVR w/ Moving User	0.9241	0.9210	0.9557

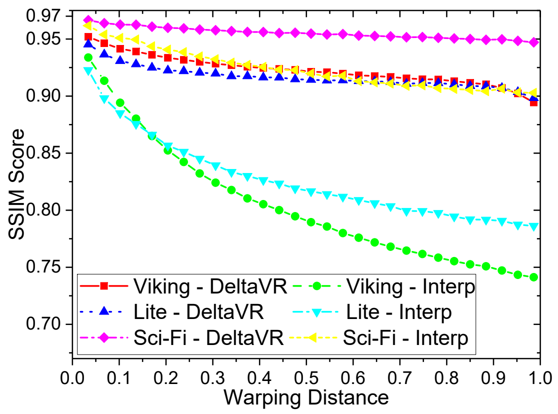


Figure 3.19: Effectiveness of delta patching

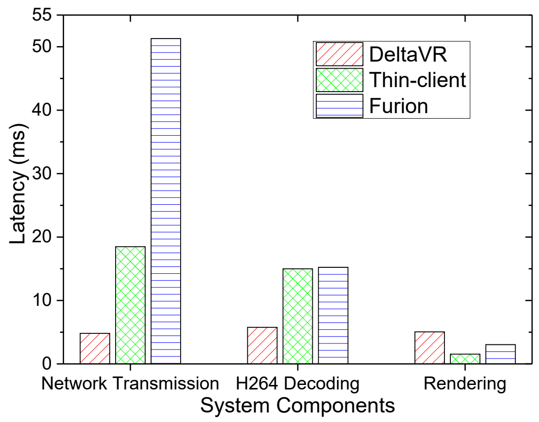


Figure 3.20: System latency of DeltaVR

Image Quality

We evaluate the VR image quality provided by DeltaVR using the SSIM metric [83], which quantifies the image quality degradation in DeltaVR from the pristine high-quality image rendered at the cloud. The results in Table 3.2 show that DeltaVR ensures high image quality (> 0.9) in all the VR applications with fast user movement, and significantly outperforms that of local frame rendering. Such improvement on image quality allows many advanced graphics options such as shadow casting and anti-aliasing at the mobile HMD, and greatly enhances the user experience.

Besides, we also evaluate the effectiveness of delta patching in DeltaVR by comparing with the traditional image warping technique that interpolates pixels in disoccluded regions [63]. Figure 3.19 shows that delta patching in DeltaVR experiences much less image quality degradation when the warping distance increases, and retains SSIM> 0.9 even when the warping distance increases to 1.0. In comparison, the VR image quality in traditional image warping quickly drops when the warping distance is larger than 0.2, because of the increased disoccluded areas in VR frames with higher VR dynamics.

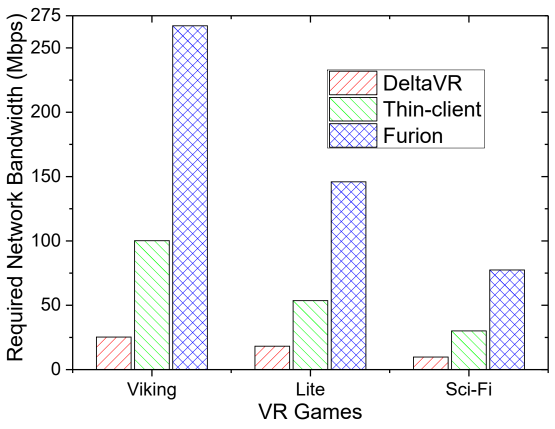


Figure 3.21: Network bandwidth required by DeltaVR

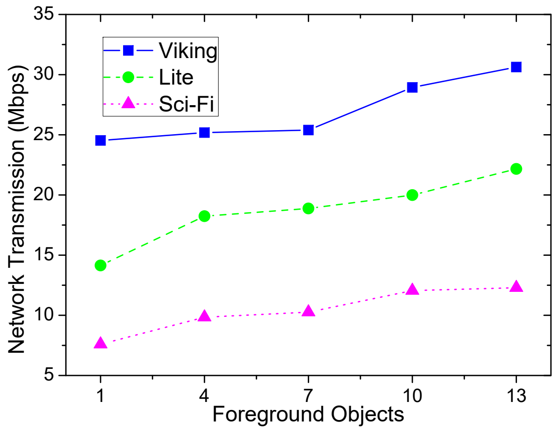


Figure 3.22: Bandwidth for different VR scenes

Latency

Motion-to-photon latency is critical in VR to ensure user experience and avoid motion sickness, and such latency depends on 1) the transmission delay of reference frames and delta images, and 2) the computation delay of frame decoding, image warping and delta patching at the mobile HMD. These delays over the Viking Village game are averaged over all the VR frames, and Figure 3.20 shows that the total processing latency for each VR frame is less than 16ms. More specifically, the transmission delay in DeltaVR is about 4.8 ms due to the minimized size of delta images, and is less than 10% of that of Furion [52] which pre-fetches the panoramic background images for all possible directions of user movement. Similarly, the H.264 decoding delay in DeltaVR is also 66% lower than the existing schemes because of the smaller amount of VR frame data being transmitted. At the mobile HMD, DeltaVR takes about 5.1 ms for frame rendering, which is slightly higher than other schemes due to the extra overhead of image warping and delta patching.

3.7.3 Impact of Wireless Connection

The condition of wireless connection between the cloud and the mobile HMD is critical to DeltaVR. Being different from Furion [52] which requires gigabit WiFi connection to transmit full VR frames, Figure 3.21 shows that DeltaVR requires at most 25 Mbps of network bandwidth, which can be easily satisfied by any existing WiFi. In addition, the required network bandwidth also depends on the VR scene complexity, and Figure 3.22 shows that DeltaVR effectively restrains the required network bandwidth within 35 Mbps over highly complicated VR scenes, which consist of more than 13 foreground objects.

3.8 Related Work

Mobile Offloading and Cloud Gaming: General-purpose mobile offloading reduces the local computational burden of mobile devices, by adaptively partitioning the computing tasks and offloads only the most appropriate portion to the cloud for remote execution [22, 35]. However, it is difficult to partition the process of rendering a VR frame, which is operated by GPU hardware. The amount of frame data sent to the mobile HMD, hence, remains unchanged.

Our proposed design of DeltaVR is related to prior work on cloud gaming [28, 74, 43]. Existing commercial systems such as PlayStation Now and NVidia Shield, consider frontend mobile devices as a thin client, to which the game's output is streamed as compressed video. However, these designs cannot scale to mobile VR, because its requirements of high resolution and low response latency make it impossible to stream game scenes at real-time. MoVR [11, 10] enables multi-Gbps wireless communication to VR headsets via mmWave wireless technology, but relies on specialized hardware support and line-of-sight connectivity. Instead, DeltaVR significantly reduces the amount of VR frame data being transmitted, and hence maximizes the mobile VR performance over conventional wireless networks.

Collaborative Rendering: Recently, researchers advocate the idea of collaborative rendering, which splits the computing workload of rendering individual frames between the cloud and local mobile devices. Flashback [16] pre-renders all the VR frames offline and caches the rendered frames at the HMDs' local storage, so as to alleviate the run-time computation burden. However, the system performance deteriorates quickly with the number of dynamic VR objects and the unavoidable cache miss. It also consumes a huge amount of storage space at mobile devices (e.g., 50GB data for each VR application). Kahawai [23] exploits the cloud GPU to render high quality images that enhance the visual quality of the locally rendered images, but still leaves heavy rendering workloads on the mobile. Furion

[52] separates the VR scenes into background and foreground layers, and streams only the panoramic background images to mobile devices, but has to speculate future user movement for prefetching and hence suffers from misprediction. Instead, DeltaVR does not involve any pre-fetching of VR frames, and is hence resistant against sporadic VR application events or user behaviors.

Graphic Processing: Image-based rendering [63] is widely used in today's VR applications, but incurs fast degradation of image quality with large warping distance due to the view disocclusion. Asynchronous TimeWarp technique [tim] in mobile VR compensates and displays the previous frame with the current head rotation when the mobile fails to render on time, but leads to flickering edges with vigorous head motions. [69] aims to mask the network latency by provisioning an extra view at deliberately selected location to fill the disoccluded holes, but requires more computations to warp the extra image. Post-processing techniques [80] interpolate or extrapolate the disoccluded view, but lead to blurry regions. In contrast, DeltaVR captures all disoccluded views in advance as the delta image, and hence guarantees high VR image quality regardless of the heterogeneous dynamics in VR applications.

Some other schemes [76, 77] propose to adapt the resolution for different parts of the panoramic images according to the user view point, so as to reduce the amount of image frame data being transmitted. These techniques, however, still transmit full VR frames and are hence susceptible to vigorous user head rotation or movement in intensive VR scenarios. In contrast, DeltaVR transmits only delta images with minimum sizes to the mobile HMD, enabling fast response to any dynamic user behavior.

3.9 Discussions

3.9.1 Supporting Next-generation VR Systems

The VR industry is rapidly evolving towards wider FOV and higher pixel density. For example, the FOV and resolution of Pimax 8K [pim] reach to 200° and 3840x2160 for each eye. Such change of image resolution quadratically increases the image size, which aggravates the insufficiency of wireless network bandwidth. However, DeltaVR is less impacted because the redundancy between VR frames persist, resulting in constantly small delta images. On the other hand, the mobile VR performance will be impaired by the computational overhead of image warping, which is proportional to the image resolution. To address this issue, we plan to adaptively adjust the density of the mesh grid in mobile IBR, so as to reduce the computational overhead of image warping.

3.9.2 User Sensitivity to VR Latency

The latency in a VR system can typically be divided into two categories. First, the motion-to-photon latency represents the elapsed time for the user head motions to be reflected on the user screen. As noted in [17], human sensory system is more sensitive to such latency, which requires to be less than 20 ms to be imperceptible. Our work reduces such latency by sampling the user pose right before mobile rendering so as to exclude the influence of the networking and decoding delay. In contrary, the latency of user interaction to the virtual environment, such as flipping the switch or pushing the controller button, is much less important and users can tolerate up to 50 ms of such latency. Therefore, the high performance in our work as shown in Figure 3.20 allows plenty of time for cloud processing and ensures robustness to large system variance without compromising the user experience, as long as each pipeline stage can finish in 16 ms.

3.9.3 Multi-user Support

Multiple VR users may execute VR applications at the same edge cloud server with finite resources, which may exceed the cloud capacity. Our future work plans to share GPU between individual VR users so as to reuse the results of cloud rendering and hence reduce the computational overhead in the cloud. It is observed that temporal and spatial locality widely exists in VR games [18] and hence the pixels in the rendered images across multiple users should be redundant as well. By reusing the pixels from other users [88, 65, 14], the cloud GPU can avoid the repetitive computations and reduce resource utilization.

3.10 Conclusion

In this chapter, we present DeltaVR, which achieves high-performance mobile VR over heterogeneous VR dynamics, by adaptively reusing the redundant VR pixels across consecutive VR frames. DeltaVR utilizes the cloud to determine the pixel redundancy between frames and transmits only the distinct portions as delta images to the mobile HMD, so as to fundamentally reduce the amount of VR frame data being transmitted without impairing VR image quality in anyway. Based on the implementation and evaluation over Android OS and Unity engine, we demonstrate that DeltaVR maximizes the mobile VR performance with 95% less amount of wireless data transmission.

Chapter 4

Interconnecting Heterogeneous Devices in the Personal Mobile Cloud

4.1 Introduction

Nowadays, a mobile user is usually equipped with multiple types of mobile computing devices, ranging from traditional smartphones and tablets to emerging wearables, each of which is designed for a specific application scenario. Such diversified designs satisfy the unique requirements of different application scenarios, but also restrict the performance or usability of these mobile devices in other aspects. For example, wearable devices enable body sensing with a small form factor, at the cost of limited capacities in computation, communication and battery life. A viable solution to eliminate such restriction is to construct a *personal mobile cloud* [31], which incorporates and interconnects all the mobile devices owned by a user via wireless links. These devices are then able to flexibly share system resources with each other, augmenting the mobile computing capability provided to the user. For example, wearables can save their local battery by exploiting the computational power of nearby stronger devices [22, 50, 75], while providing their sensory data to these devices and facilitate their context-aware applications [41, 82].

The major challenge of realizing such a personal mobile cloud is the heterogeneity of mobile computing devices, which resides in both hardware and software aspects and prevents these devices from being interconnected in a generic manner. First, the increasing variety of hardware components being mounted on today's mobile devices results in fundamental difference in the drivers, I/O stacks and data access interfaces being used by these hardware. Even for the same type of hardware, access to the hardware data from a remote system could fail if the hardware drivers are provided by different manufacturers and incompatible with each other. Such incompatibility is usually a result of customized SoC designs adopted by different hardware manufacturers. For example, the accelerometer drivers for the Qualcomm Snapdragon chipsets are definitely incompatible with the Samsung Exynos chipsets. Second, the complexity of today's mobile applications has been dramatically increased, leading to heterogeneity in both their requested types of mobile system resources and their specific ways of accessing these resources. Existing solutions, unfortunately, are limited to interconnecting mobile devices with respect to an individual mobile application [60, 86] or a specific type of shared hardware [15, 72]. Therefore, they will need a large amount of reprogramming efforts to interconnect heterogeneous mobile devices, by rewinding the wheel for each individual hardware or software component of these devices. Such reprogramming efforts do not only impair the usability of mobile computing system in versatile environments, but also incur additional overhead to the operation of mobile OS and hence reduce the mobile system performance.

The key to generic interconnection across heterogeneous mobile devices is to develop an efficient framework for the sharing of peripheral resources between these devices, which appropriately masks the hardware and software heterogeneity in mobile systems from each other. Development of such a peripheral sharing framework, however, is challenging due to the close interaction between mobile hardware and software. A framework at the lower layer of mobile OS hierarchy unifies the heterogeneous peripheral requests of mobile applications, but has to tackle with individual hardware drivers which are operated in intrinsically different ways [12] and hence incurs a tremendous amount of re-engineering efforts. Sharing system peripherals at the application layer, on the other hand, is able to access mobile hardware through a generic OS interface, but has to be associated with specific data transfer protocols and hence has limited generality [29, 39].

In this chapter, we present a mobile system framework to address the above challenges and generically interconnect heterogeneous mobile devices towards a personal mobile cloud.

Our basic idea is to develop the resource sharing framework as a middleware in the mobile OS, which exploits the existing mobile OS services to share peripherals between mobile devices. These services hide the low-layer details of device driver operations while providing unified data access APIs to user applications. Interconnection between mobile devices, then, could be realized via remote access and invocation of these OS services. Since these services are executed as a standalone system process by the OS kernel and are separated from application processes, remote service invocation can be done via inter-process communication (IPC) between mobile systems without involving complicated issues such as memory referencing and synchronization. As a result, any new device can be incorporated into the mobile cloud by inserting our framework into its OS, without modifying the OS kernel, our framework itself, or the source code of any mobile application.

We have implemented our design on Android OS with less than 5,000 Lines of Codes (LoC) over various mobile platforms including smartphones, tablets and smartwatches, and demonstrated the efficiency of sharing various types of hardware (GPS, accelerometer, audio speaker, camera) between remote mobile devices. The evaluation results show that our design can efficiently support ubiquitous access to system peripherals between remote systems with arbitrary mobile applications, without incurring any significant system overhead.

The rest of this chapter is organized as follows. In Section 4.2 we provide a high-level overview about our motivation and designs. Section 4.3 and Section 4.4 present the details of our peripheral sharing framework and application interface. Section 4.5 describes how we support sharing multimedia peripherals between mobile devices. Section 4.6 presents our implementations over various mobile platforms, and Section 4.7 presents our evaluation results based on these implementations. Section 4.8 discusses the related work. Finally, Section 4.9 discusses and Section 4.10 concludes the paper.

4.2 Overview

In this section, we start with a brief description about the layered architecture of mobile OSes, which motivates our proposed design. Based on this motivation, we further provide a high-level overview of our proposed framework.

4.2.1 Motivation

Our design is motivated by the layered architecture of mobile OSes, as discussed in section 1.2. In such layered architecture, peripheral resource access is provided by system services via a suite of generic and pre-defined APIs, which are invoked by user applications via IPC with binder mechanism in Android and message passing in iOS, respectively. For example, instead of directly accessing GPS or WiFi network interface, an application in both Android and iOS retrieves the location information of the device via a location service provided by the OS.

We support generic peripheral sharing between remote mobile devices based on the such layered architecture of mobile OSes. First, since system services are the only interface for user applications to access system peripherals, access to any type of peripherals on a remote device could be provided by the same generic framework, as long as this framework can intercept the requests of peripheral access from user applications and redirect these requests to the remote system. Furthermore, different types of system services are invoked following the same mechanism (e.g., binder in Android and message passing in iOS), and hence we will not confront with the heterogeneity of service operations. Second, these system services hide the details of hardware operations from user applications. Hence, peripheral sharing based on these services addresses the heterogeneity of hardware driver implementations, and allows devices with different hardware models and drivers to access each other. More importantly, since system services are invoked through the pre-defined set of APIs, interception and redirection of these invocations are transparent to user applications, which will access the remote system peripherals in the same way as they access the local counterparts, without any modification to their source codes.

4.2.2 The Big Picture

As shown in Figure 4.1, our proposed middleware resides between user applications and OS services, and consists of two major components: *a)* Application Interface and *b)* Peripheral Sharing Framework.

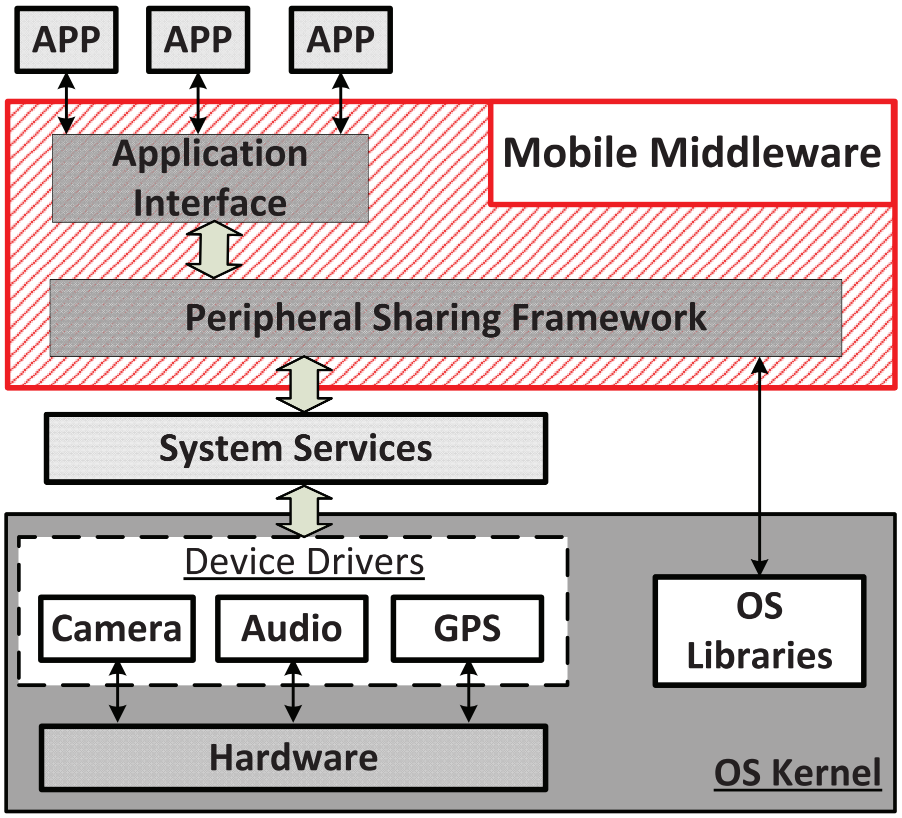


Figure 4.1: The big picture of interconnecting heterogeneous mobile devices

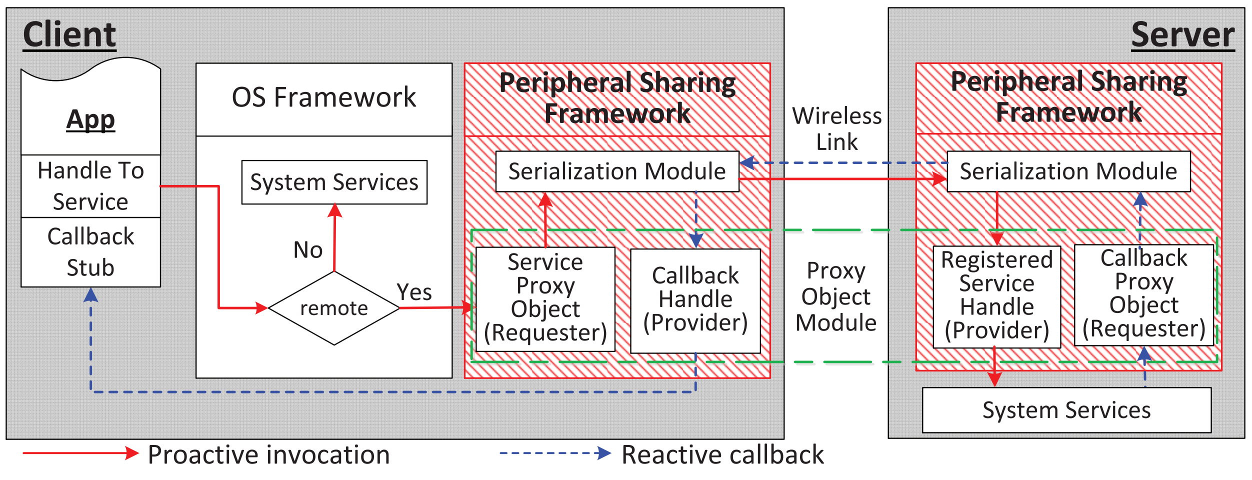


Figure 4.2: Design of peripheral sharing framework

The *Application Interface* regulates how a user application accesses the shared peripherals at the remote system through an application-specific metadata file that configures the usage of remote resource. When a user application requests to access a system peripheral, the Application Interface parses the configurations to return the appropriate handle to the corresponding system service. Hence, no matter what system service is invoked and whether the service is invoked at the local device or the remote device, the service is operated through the same way.

The *Peripheral Sharing Framework* interacts with the local OS and communicates with the remote OS services to provide remote peripheral access to user applications. As shown in Figure 4.1, the details of low-layer device driver implementations and hardware operations in the OS kernel are completely separated from the peripheral sharing framework, and different OS services are invoked in a universal manner through the same set of pre-defined APIs.

4.3 Peripheral Sharing Framework

Our design of the peripheral sharing framework is shown in Figure 4.2. In general, our framework intercepts the requests of peripheral access generated from local mobile applications, and forwards these requests to another remote mobile device which acts as the server and provides the shared resource. Every time when a peripheral access request is received, the server will invoke its local OS service corresponding to the requested peripheral resource, and reply with the resource data.

This framework consists of two major components: *Proxy Object* module and *Serialization* module. The responsibility of the Proxy Object module is to serve as a portal of remote service invocation, and manage the proxy objects for peripheral sharing at both endpoints of the peripheral sharing system. The Serialization module is responsible for data serialization, which is the process of converting memory objects into a binary format that can be transmitted through the network link. These binaries can be reconstructed back to memory objects by deserialization at the other endpoint.

Our framework supports peripheral resource access between mobile devices in two ways: *proactive invocation* and *reactive callback*. First, when a remote system service is available, a service proxy object will be created by our framework to initiate the IPC between the client and the server. In proactive invocation, every time when an application requests to remote service access, the proxy object at the client triggers a remote invocation event, which is captured at the server to invoke the corresponding service method. Second, applications can also access system peripherals reactively by receiving data in system events, e.g., location update. Peripheral resource access in this case is handled by reactive callbacks, which allow applications to register and listen to a system event with a callback handle. This handle is called via a callback proxy by the service at the server when the system event occurs, and then used to transmit resource data back to the client. This invocation procedure is similar to proactive invocation, but in a reverse direction.

4.3.1 Invocation of Remote OS Services

Our framework supports remote invocation of both Java-based and native OS services. As we mentioned above, both types of services can be remotely invoked via both proactive invocation and reactive callback.

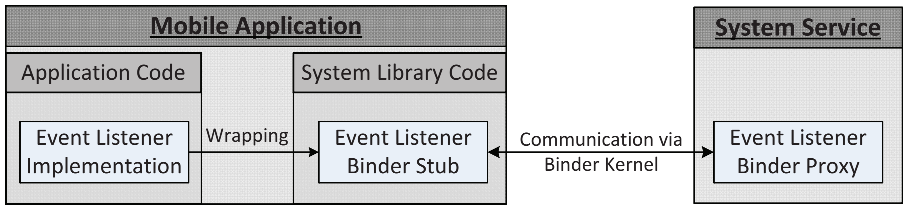

Mobile Application
Application Code
System Library Code
Event Listener Implementation
Wrapping
Event Listener Binder Stub
Communication via Binder Kernel
System Service
Event Listener Binder Proxy

Figure 4.3: Resource sharing through callbacks

Java-based OS Services

In Android, part of system services are implemented in Java and running in a standalone system process. On one hand, to devise a generic solution to the serialization module for sharing these services, we utilize the *Java reflection mechanism* which enables developers to inspect classes, interfaces, fields and methods at run-time without knowing the names of classes and methods in advance. Specifically, we access all the field values of any memory object by reflection and convert them into binaries. Similarly, this feature can be applied for deserialization by creating the object instance and setting all the field values at run-time, so as to reconstruct memory objects from the received binaries.

On the other hand, we develop the proxy object from the existing system service class, by appending the bytecodes of remote invocation operations to the service class via *dynamic weaving*. The dynamic weaving technique in Java allows us to instrument the bytecodes of an existing Java class at run-time, generating a new class instance as the subclass of the original class type. Since a system service class in Android is specified as a subclass of the Binder class by itself, we dynamically weave a system service class at run-time whenever it will be remotely accessed, with the extra bytecodes of remote invocation operations added to service methods. Hence, this newly weaved class will be a subclass of the Android Binder class and can be registered to binder kernel driver to receive and intercept the applications' invocation to a service method.

Our framework also supports callback to user applications which register a system event with any class object implementing the event listener interface. Then, as shown in Figure 4.3, user applications exploit the Android system library to wrap this listener into a binder stub which communicates with the binder proxy for event listening in the corresponding system service. To realize such callbacks across two mobile devices, as shown in Figure

4.2, we allow an application to register and listen to an event in the remote system via a *callback proxy*, and utilize the event listener binder proxy at the client as the *callback handle*. Afterwards, when the system event happens at the server, the callback proxy will initiate a remote invocation to the callback handle at the client.

Native OS Services

Another large body of system services in existing mobile OSes is implemented in native C/C++ languages, e.g., sensor and graphic services in Android and all system services in iOS. Since these services are executed as compiled binaries at run-time, proxy objects for these services cannot be developed through run-time manipulation due to the following reasons. First, it is hard to locate the entry and exit points of a native method at run-time, and hence difficult to dynamically attach the weaving instructions to the service class. Second, the machine instructions in these native service classes depend on the hardware architecture and hence can only be operated and compiled statically.

To address this challenge, in our current design we modify the source codes of each system service class to realize the functionality of serialization and deserialization, as well as the remote invocation of service methods. Being different from existing schemes which share system resources over the device drivers and have to reprogram the driver implementations for each individual system, our modifications are applied to OS services and applicable to all mobile systems with heterogeneous hardware components. This advantage enables our work to be applied to a variety of different mobile platforms, and we will describe such implementation details in Section 4.6.

4.3.2 Unix Domain Socket

Another IPC scheme supported in our framework is *Unix domain socket*. For example in Android, sensor data is not delivered from the sensor service to applications as method arguments of the binder method invocation. Instead, it is delivered by a Unix domain socket between the sensor service and the application, in order to reduce the system overhead of highly frequent data transmission.

In our design, we build the reliable data exchange channel between mobile devices with a network socket instead of the Unix domain socket used in existing IPC schemes. More specifically, the system service process in the server will listen to the connection request for building a distributed data channel. When an application requests a remote system service, the proxy object in the client will establish a connection to the server and pass the file descriptor of this connection back to application. Thereafter, the system service in the server can exchange data reliably with the client application through TCP. Since the mobile OSes operate these two types of sockets with the same APIs, the actual type of the data channel socket can be hidden from the system which uses the data channel and the existing IPC code for data exchange can be kept unchanged.

4.4 Application Interface

Our design of the Application Interface is shown in Figure 4.4. Whenever a user application requests to access an OS service, the Application Interface intercepts this request and returns a handle to the appropriate service, which could be located in either the local system or the remote system. Since both handles provide the same programming interface to applications, the process of peripheral resource sharing between mobile devices is completely transparent to user applications.

Decisions on the service handle being returned are made based on the configurations stored in an application-specific *Metadata File* in the application directory. More specifically, when a user application requests to access a system service, the framework loads the configurations from the metadata file. If the configurations indicate that a local system peripheral will be accessed, the local service handle is returned to the user. Otherwise, our framework will create a remote service proxy and build a TCP connection to the remote system for peripheral access. In our design, this metadata file is operated by a special *Proxy Application* which is embedded as part of the Application Interface. This proxy application manages and overrides the peripheral access configurations for all user applications, and also receives inputs from the mobile OS settings about the list of available system resources. All

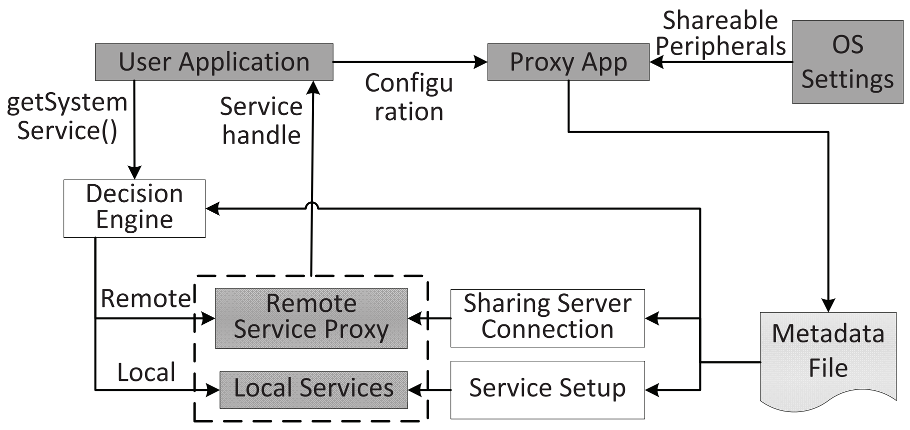


Figure 4.4: Design of application interface

these information will be written by the proxy application into the metadata file, which are then checked by our framework at run-time to ensure correct service invocations.

Through the development of this proxy application, our design allows a user application to configure its access of system peripherals in three ways. First, we allow developers to distribute their configurations along with the application binaries during installation, and explicitly specify how the application will access system peripherals. For example, the developers can decide the destination of the remote peripheral and the data rate at which they want the remote peripheral to be accessed. Second, if the target for peripheral resource sharing is unknown, developers can opt to adopt existing service discovery protocols (e.g., [89, 73]) and explore for the available shared peripherals nearby, by specifying the service discovery protocol being used in configurations. Third, we also allow mobile users to manually modify the configurations of peripheral resource sharing via the proxy application. For example, a mobile user can explicitly configure the application to project the screen content to a nearby large LCD.

4.5 Supporting Multimedia Operations

Operations over multimedia peripherals usually involve large sizes of bulk data and need to exploit shared memory for data exchange between applications, resulting in new challenges when sharing these peripherals between remote mobile devices. In this section, we present our design to support the access to such shared memory of multimedia services between

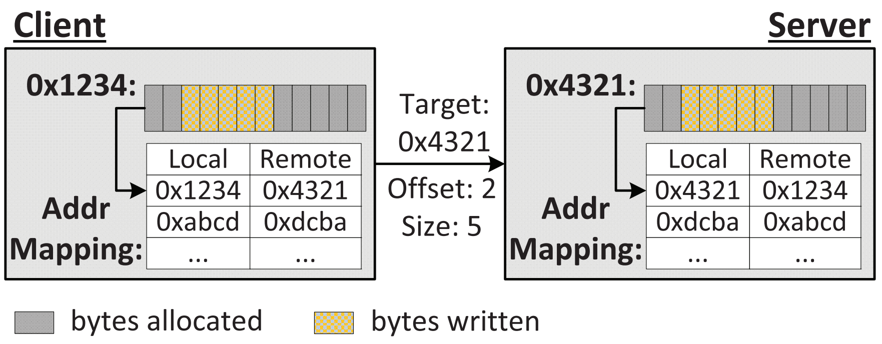


Figure 4.5: Memory synchronization for content part

remote mobile devices. The major challenge, however, lies in how to ensure the remote IPC reliability with the minimum intrusion to the original mobile OS structure and interface.

According to the way shared memory is operated, we categorize the shared memory into two cases and handle them separately: the general buffer and the graphic buffer. The general buffer is defined as shared memory that is portable and vendor-independent. It is directly allocated and operated by system services. The graphic buffer, on the other hand, stores image data such as camera preview and video frames, and is usually operated by vendor-specific HAL.

4.5.1 The General Buffer

The general buffer can be divided into two parts. First, the *content part* stores the resource data and is operated following the producer-consumer pattern, i.e., one operator always writes data into the buffer and the other always reads the data. In this way, the two operating parties are always synchronized without contentions on writing. Second, the *control part* stores the control information that is necessary to operate the content part, such as the reading and writing pointers. The control part, therefore, can be written by both applications and system services which may conflict with each other when writing. In our design, we develop different techniques for synchronizing the shared memory of these two parts.

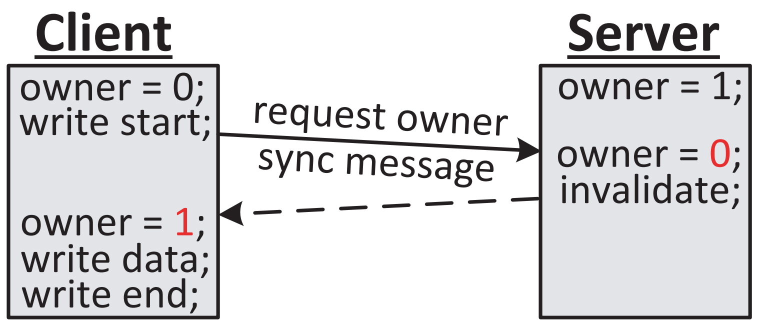


Figure 4.6: Memory synchronization for control part

Content Part

To efficiently operate the content part, being different from traditional approaches of Distributed Shared Memory (DSM) [46] which always shares memory in fixed-size units and may result in large amounts of redundant data synchronization between mobile devices, we flexibly synchronize the shared memory at arbitrary sizes based on the actual application patterns of memory access. This flexibility is mainly due to the fact that there is no write contention between the sharing client and server when they synchronize the content part, ensuring any size of synchronized memory to be always coherent. As a result, our basic idea is to establish a memory mapping between the client and the server, and synchronize the memory contents based on the mapping. Whenever one endpoint allocates a block of shared memory, a corresponding memory block with the same size is allocated correspondingly at the other endpoint, and a mapping entry between their addresses is added into the mapping table maintained at both endpoints. Whenever one endpoint finishes its write operation, we use the mapping entry and the writing offset to calculate the destination writing address and synchronize the data being written. For example in Figure 4.5, when the client allocates 12 bytes of memory, the server also allocates the same amount of memory accordingly, and the addresses of both the client and server memory are stored in the mapping table in both endpoints. Later, when the client writes 5 bytes into the buffer, the resource sharing framework synchronizes and updates memory with the appropriate size, according to the received target address and offset.

Control Part

We also develop a flexible synchronization scheme for the control part, which can synchronize either the whole control part or the individual fields of it. To ensure data consistency with write contention, we establish a happened-before relation between two mobile devices through an ownership flag, and only allow the owner of a memory field to write to the field. The ownership transfers when the other endpoint tries to write the field. In this way, we ensure that writes happen in sequential turns between the client and the server and hence the memory at two endpoints will always be consistent. For example in Figure 4.6, the client intends to write a control field but does not have the ownership to the field. Therefore, the client sends a message to the server and requests for the ownership of the field. Having received this message, the server transfers its ownership to the client.

4.5.2 Graphic Buffer

Graphic services in the mobile OS, which operate multimedia devices such as the camera or LCD screen, usually utilize the GPU to accelerate the speed of image processing and rendering. Hence, being different to the general buffer which is allocated and written by the applications or system services themselves, the graphic buffer are allocated and operated by the vendor-specific HAL. As a result, we cannot directly intercept the buffer operations from our peripheral sharing framework and further establish memory mapping for synchronization between remote systems. Instead, our approach is to integrate our peripheral sharing framework with the APIs provided by the OS for user applications to manage and operate the graphic buffer.

We use Android OS as a nominal example to present the details of our design. In Android, the core of its graphic services is the *BufferQueue* class, which manages different graphic buffers allocated by the vendor-specific *gralloc* module. The operation of graphic buffers also follows the producer-consumer pattern: the producer (usually the hardware device driver) dequeues an empty buffer handle from BufferQueue and queues the filled buffer back to BufferQueue; the consumer (e.g., Surface Flinger) acquires a handle of filled buffer from BufferQueue and releases the consumed buffer back.

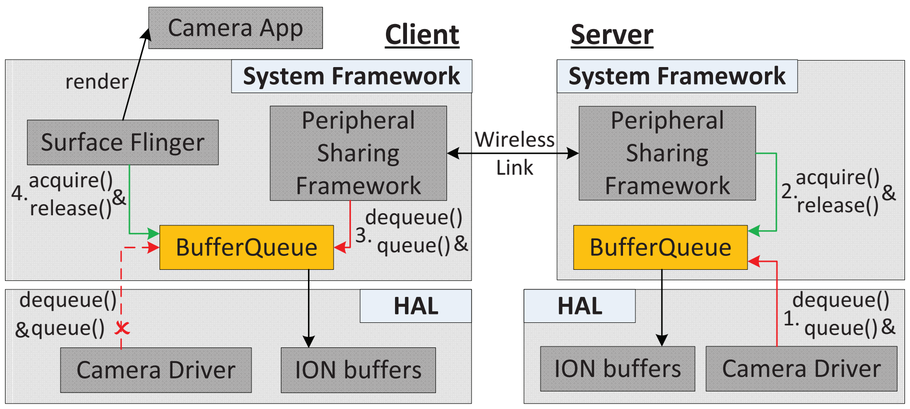


Figure 4.7: Operating the graphic buffer for remote camera access

As a result, our resource sharing framework acts as a consumer of BufferQueue to extract the graphic buffer contents, and then pushes these contents to the remote device. For example in Figure 4.7, after the camera driver in the server posts a preview image buffer into the BufferQueue, our framework collects the graphic buffer contents as a consumer and sends them to the client. Then, the framework in the client retrieves an empty buffer, fills the buffer with the received image content and posts the buffer back into BufferQueue. Afterwards, the Surface Flinger in the client renders the preview image.

4.6 Implementation

We implemented our design in Android v5.1.1 with CyanogenMod 12.1, and build the peripheral sharing framework on Linux distribution Ubuntu 12.04. The dynamic weaving technique is implemented with dexmaker[1]. Our implementation consists approximately 1,500 lines of Java code and 1,850 lines of C++ code to support Java-based and native system services, respectively. It is deployed on multiple types of mobile devices, including smartphones, tablets and smartwatches.

Based on this implementation, we are able to further implement the functionality of sharing different types of resources between remote mobile devices, and the implementation details are listed in Table 4.1. First, our framework supports remote access of the location service, which involves operations of both GPS and WiFi, with less than 10 Lines of Java

[1]https://github.com/crittercism/dexmaker/

Table 4.1: List of Supported Services

Service	Type of code	LoC	Hardware
Location	Java	<10	GPS, WiFi network
Sensor	C++	283	All onboard sensors
Audio	C++	647	Speaker
Camera	C++	594	Camera

code. Comparatively, remote access of other system services involves operations over native classes and requires more LoC on data serialization and deserialization.

4.6.1 Service-Specific Optimization

Sensor Service: We optimized the sharing of sensor service and allow mobile applications to receive sensor data at their specified rates, no matter how fast such data is generated by the hardware or OS, so as to minimize the data transmission cost of ubiquitous sensor access. More specifically, we attached a specialized control module to the socket channel for sensor data exchange between mobile devices, and customized the data transmission rate between mobile systems without modifying the sensor service methods or the sharing framework themselves. In practice, the sensor service in the server will receive the sensor data rate from the client, and send sensor data to the client only if necessary.

Audio Service: In Android, the audio playback will not start until the audio buffer is fully filled. However, the amount of audio data that an application writes in one operation may not be enough to fill up the buffer. As a result, mobile users may experience a long latency of initializing remote audio playback if the framework synchronizes as soon as a write operation happens. In order to reduce such latency, our framework accumulates the buffer contents and holds from synchronization until the local buffer is fully filled. Consequently, we eliminate the multiple round trips for the initial buffer synchronization of audio playback.

Notification Service: Besides the system services operating the hardware resources, another collection of system services also exists to let mobile applications utilize the system-wide software resources. Since software system services interact with mobile applications in the same way as hardware system services, our framework also supports sharing of software system services between mobile systems. We have implemented the sharing of Android

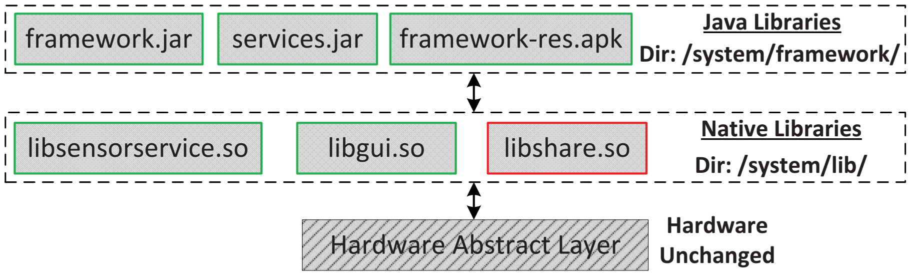


Figure 4.8: Porting sensor service libraries in Android wear

notification service between mobile systems with less than 15 lines of Java code, and allow the user to show notifications on a remote mobile device. When a mobile user clicks the notification icon, the action associated with this notification will be performed back to the local mobile device through remote callback.

4.6.2 Deployment over Different Mobile Platforms

Our proposed framework allows generic peripheral sharing among heterogeneous types of mobile devices, which are equipped with hardware from different manufacturers or running different versions of device drivers. Being different from traditional DSM-based schemes which have to manually port and reprogram the driver implementations from one device to another, our framework utilizes the OS service interfaces to hide the hardware heterogeneity from user applications and realizes automated migration between mobile platforms without manual modification. In our implementation, we share system peripherals among smartphones, tablets and smartwatches, even if their implementations of hardware drivers are not open-sourced and not accessible in CyanogenMod. Instead, we exploit the source code of the Android framework service layer, which is publicly available in the Android Open Source Project (AOSP) and remains the same for different platforms.

Deployment of our framework on smartphones and tablets is trivial since they are directly supported by Android CyanogenMod OS. Their deployments can be simply done by building and flashing their full OS installation packages. Deployment over smartwatches, however, is more complicated because the open-sourced OS codes for Android wear is incomplete. We deploy our framework over smartwatches by porting and replacing the existing library files in the rooted smartwatch. Since these libraries are dynamically loaded and linked with interface symbol names in Android, such library replacement will not affect OS execution as long as the new libraries keep the same programming interfaces as the old ones.

For example, for the Android sensor service shown in Figure 4.8, we build individual modified modules as library files. More specifically, the Java libraries serve as the entry for user applications to interact with the native sensor service, and hence are modified to return the remote service handle to applications. The native libraries receive the service invocation from Java libraries and request the local or shared peripherals. These individual library files are then used to replace the files with the same names in the directories on a rooted device. Note that, this porting technique is generic and applicable to all other system services such as location service and multimedia services, because the source code of these system services are also available in AOSP and the service libraries are dynamically linked and loaded.

4.7 Performance Evaluation

In this section, we evaluate the performance of our proposed designs on sharing peripherals between remote mobile devices. More specifically, we first evaluate the general performance of peripheral sharing between remote systems by adopting resource-specific performance metrics, and show that our design reaches satisfiable sharing performance with little computational overhead. Afterwards, we evaluate the power consumption of peripheral sharing between remote mobile devices, and demonstrate that peripherals at remote mobile devices can be accessed without consuming significant amounts of energy. Last, we measure the network throughput of sharing different types of peripherals, and report the amount of wireless network bandwidth required to support resource sharing. Our experiments are performed by sharing GPS, accelerometer sensor, speaker and camera between mobile devices. Note that our evaluations are directly performed over individual hardware components, which are accessed by the mobile OS itself. On the other hand, since our proposed peripheral sharing framework keeps the method of user applications' peripheral access as intact, it is able to seamlessly support any off-the-shelf mobile application with the corresponding modification of the resource metadata file.

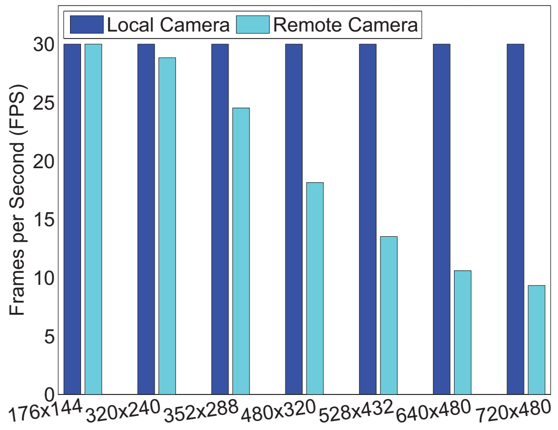


Figure 4.9: FPS for real-time camera preview

4.7.1 Experiment Setup

We perform our experiments over different types of mobile platforms including Samsung Galaxy S4 smartphone, LG Nexus 4 smartphone, Samsung Nexus 10 tablet and LG Watch Urbane, all of which are running Android v5.1.1. The generality of our peripheral sharing framework then ensures its reliable execution over these mobile platforms and seamless interaction between different mobile devices. These devices are interconnected via 40Mbps campus WiFi unless explicitly stated in the paper. Our devices are placed close to each other and the network latency is about 3.5 ms. We use a Monsoon power monitor[2] to gather the real-time information about the devices' power consumption. Note that, although in our experiments only one client device is connected to the sharing server, our framework allows multiple clients to be connected to a server simultaneously.

In each experiment, we adjust the parameters of system peripherals to evaluate the peripheral sharing performance in different application scenarios. More specifically, we vary the data rates from GPS and accelerometer to emulate the requirements from different mobile applications. We play music with different audio rates which determine the amount of data being transmitted. We also share camera previews with different image resolutions, which significantly impact the data size of the preview image.

[2]https://www.msoon.com/LabEquipment/PowerMonitor/

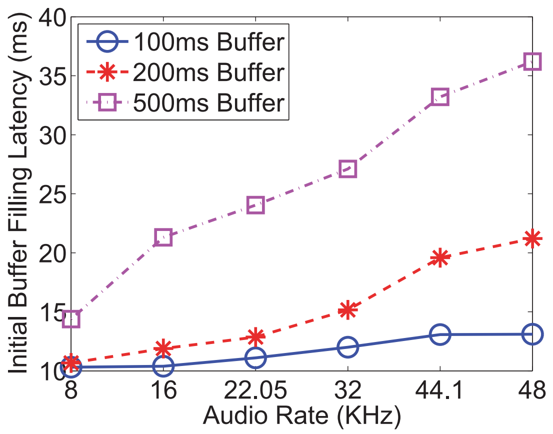


Figure 4.10: Latency of initial audio buffer filling

4.7.2 Performance of Peripheral Sharing

We first evaluate the access latency of sharing different types of system peripherals, which is measured by the average elapsed time from the time when the framework starts to process data to the time when the resource data is returned back to the local device. Such latency, hence, consists of the network transmission latency, the execution time of system service methods, and the overhead incurred by our sharing framework. Our experiments use GPS data report interval as 1 second, accelerometer data report interval as 20 ms, audio rate of 44.1 KHz and camera preview resolution of 176×144. Each experiment runs 3000 times, based on which the average data access latency is measured.

The experimental results when sharing peripheral resources between two Nexus 4 phones are shown in Figure 4.11. We can see that remote peripheral access only incurs negligible latency, which is mainly dominated by the network latency in most cases. Specifically, the network latency for GPS and accelerometer is small because of the small size of resource data being transmitted, and sharing the camera between mobile devices experiences larger network latency due to the large data size of multimedia content. On the other hand, execution of our peripheral sharing framework only incurs negligible computational overhead to mobile systems at both endpoints.

Furthermore, as we mentioned in Section 4.6.1, when we share the audio between mobile systems, the audio playback will not start until its buffer is all filled with audio data. Therefore, the latency of initial buffer filling is a key factor of users' conceived delay of using

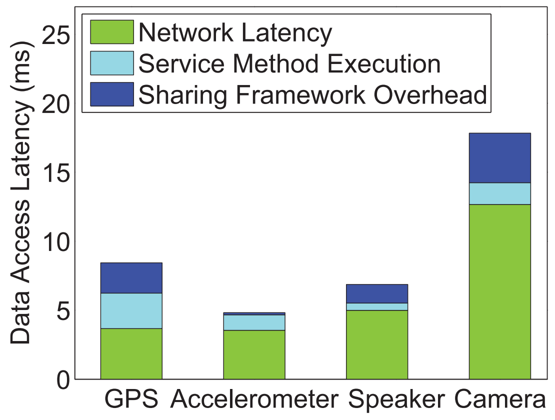


Figure 4.11: Latency of accessing shared resources

the remote speaker. Our experiments evaluate such latency by measuring the average time it takes to fill the audio buffer since the client starts to write audio data. The experiments set the buffer size as the length of audio segments with respect to the audio rate. Each experiment plays 20 audio tracks. The experimental results are shown in Figure 4.10. We can see that the initial latency of buffer filling increases along with the buffer size, but is efficiently controlled within 40ms in all cases.

We evaluate the performance of sharing the camera between mobile devices using the average frames per second (FPS) for the camera preview. Our experiments are performed with Galaxy S4 smartphones with different resolutions of camera preview over 2000 frames. From the experimental results in Figure 4.9, we can see that our framework can reach the same FPS of remote camera preview as that of the local camera, when a low resolution of 176×144 is used. Even if we increase the resolution to 480×320, our framework can still provide a FPS of 18, which is more than sufficient to support smooth camera preview (minimum FPS of 15). When the resolution further increases, the FPS will drop due to the increasing amount of data being transmitted. For example, one 720×480 preview frame has the size of 518 KB with the NV21 pixel format, hence requiring 62 Mbps of wireless network bandwidth to reach 15 FPS at the remote system.

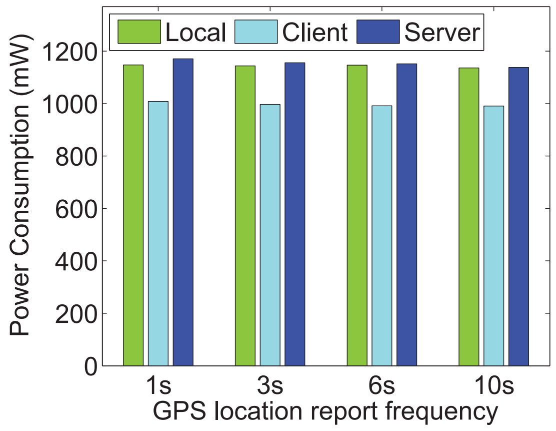


Figure 4.12: Power consumption for GPS sharing

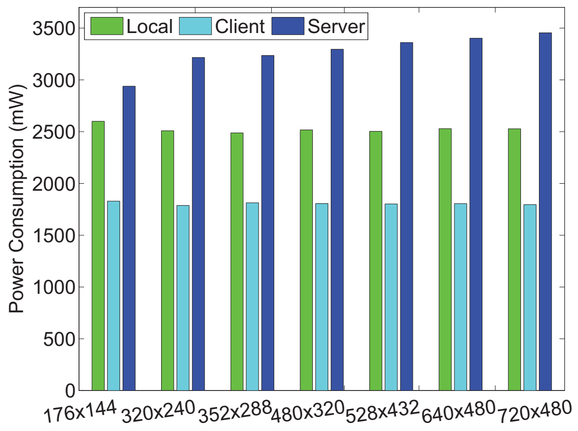


Figure 4.13: Power consumption for camera sharing

4.7.3 Power Consumption

In this section, we evaluate the energy efficiency of our work, by measuring the average power consumption at both the sharing server and the sharing client, and compare such power consumption to the power consumption of using local peripherals. To remove the dynamic power consumed by the smartphone screen, we disable the functionality of automatic brightness adjustment during our experiments and keep the display dimmest. In each experiment, we use the device for three minutes and measure its average power consumption. Each experiment runs three times over Galaxy S4 phones that are interconnected via Bluetooth links.

The experimental results for sharing the GPS and camera are shown in Figure 4.12 and Figure 4.13 respectively. In contrast to Rio [12] which consumes a tremendous amount of additional energy for resource sharing between mobile system, our framework reduces the power consumption by 13% and 29% respectively, when accessing the remote peripherals instead of the local counterparts. On the other hand, the server consumes extra energy to provide the shared peripherals, and the majority of such extra energy is consumed by sending the resource data to the client. Considering that the server is usually the device with stronger capabilities, such additional cost could be acceptable in most cases.

The experimental results for sharing the accelerometer and audio speaker are shown in Figure 4.14. From the figure we can see that the client consumes a small amount of extra power (7% and 4% for accelerometer and speaker, respectively) to access the remote

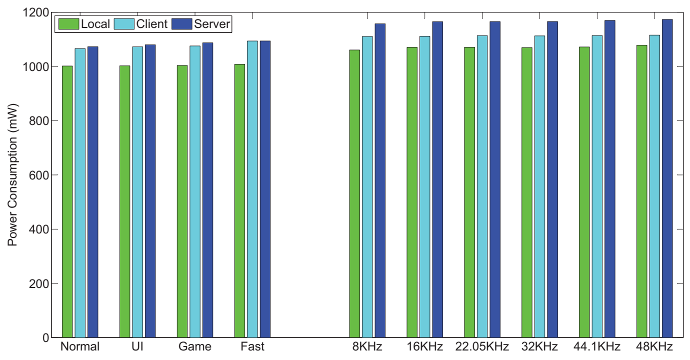


Figure 4.14: Power consumption for sharing the accelerometer and speaker

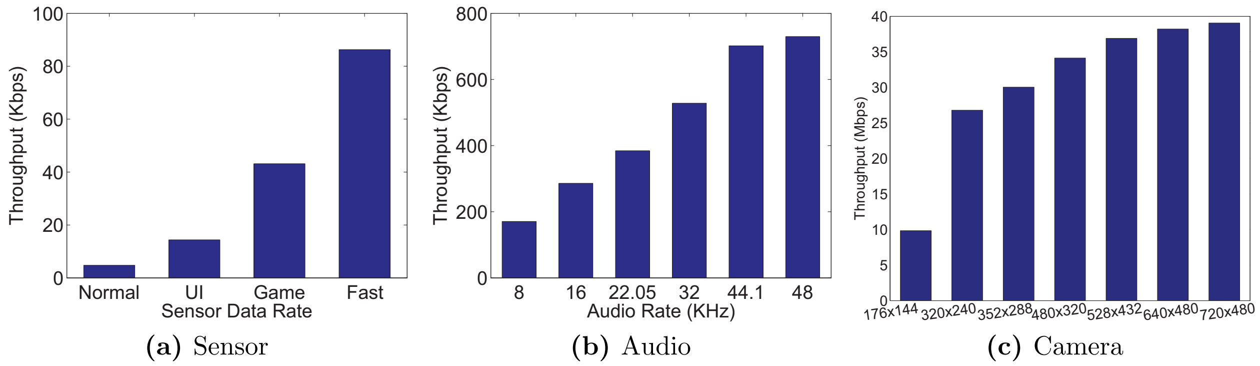


Figure 4.15: Wireless transmission throughput with remote resource sharing

peripheral. The basic reason for such additional power consumption is that both of these two resource modules are power efficient but require highly frequent synchronization of resource data, leading to additional energy consumed by wireless data transmission. In particular, adopting a higher audio rate does not noticeably increase the power consumption, because it only leads to moderate change on the size of audio data.

4.7.4 Wireless Transmission Throughput

In this section, we evaluate the amount of wireless transmission throughput being produced by the remote peripheral sharing in our framework, by measuring the average amount of data being transmitted between the client and the server. In each experiment, we use the device for three minutes to synchronize accelerometer data, play audio tracks and transmit camera previews between two mobile devices.

The experimental results for accelerometer, speaker and camera are shown in Figure 4.15a, Figure 4.15b and Figure 4.15c, respectively. We can see from the figures that sharing sensors, audio devices and camera devices incur small, moderate and large amount of wireless transmission throughput, respectively. In specific, both sensors and audio devices require only less than 1 Mbps to fully support remote resource access. Therefore, any high-speed wireless network can easily meet such throughput requirement [59]. However, remote access to the camera requires much higher wireless network bandwidth due to the large size of preview images. Therefore, the bandwidth becomes the performance bottleneck of the remote camera access, especially when a high-resolution preview is applied. Efficient network scheduling protocols are a viable solution to this bottleneck [58].

4.8 Related Work

Initial research efforts on resource sharing between systems focus on thin client, which allows clients to render graphical interfaces from and send user inputs to the server [15]. However, these systems are limited to solely sharing the graphical interface. Later on, applications have been developed to share different types of system hardware resources such as the microphone, webcam, GPS and computation [55, 78]. However, each application can only share a specific type of pre-designated hardware. Sharing any other type of resource requires a significant amount of engineering work. In contrast, our proposed framework can share heterogeneous types of system resources without incurring any reprogramming efforts.

Traditional work has been focusing on resource sharing in distributed systems. ErdOS [79] exploits opportunistic access to resources in nearby devices to efficiently save local energy consumption. In addition, resource sharing enables a device to utilize its missing resource features so as to be more powerful. Resource sharing among distributed clients has also been supported at the OS layer. Mobile grid computing [57] allows mobile devices to join the grid and share their hardware resources to other devices in the grid. However, these shared resources can only be accessed through grid-specific API. Ubiquitous computing [36, 41] enhances the performance of a system task by sharing and utilizing the resources available in the network. However, these schemes lack generality and are limited to specific applications.

Rio [12] is the first systematic solution to share multiple types of hardware among mobile systems without modifying user applications. However, its implementation over a mobile system has to closely bind with the system hardware drivers, and hence has to be reprogrammed to support different models of hardware. Furthermore, it can only provide remote resource access following pre-designated configurations, and is hence incapable of adapting to the actual resource needs of mobile applications. In contrast, our scheme, which is implemented in a higher layer in the OS architecture, is able to take the run-time application behaviors into account and leave the inconsistency of hardware drivers being dealt by the OS kernel.

4.9 Discussions

4.9.1 Pixel Format for Remote Camera Sharing

A particular issue in sharing graphic devices, such as camera or LCD display, is the consistency and compatibility of the pixel format between mobile devices. In specific, device vendors may define their own pixel format of graphic contents, and hence the graphic data may not be renderable by another device. For example, the preview data generated by the Samsung Galaxy S4 smartphone cannot be directly rendered by a LG Nexus 4 smartphone. The fundamental solution to this issue is to apply a graphic format that is compatible at all types of mobile devices, so that heterogeneous formats of the graphic pixels can be handled in a generic way. For example, the H.264 standard is used as the intermediate data format for memory synchronization by Miracast [Mir]. Instead of sending the raw graphic contents, the mobile OS first encodes these contents with H.264 codecs. Then, the H.264 data is decoded in the client and rendered to display the graphics on the screen. We will explore the possibility of incorporating such graphic data encoding into our framework in the future.

4.9.2 Access Control

An authentication or access control scheme is necessary among mobile systems to protect them against malicious parties which may send mobile malware along with the data sharing traffic or steal private information from users. Such access control can be supported in our framework by adding a new authentication layer before serialization. Various user or device identities, which are not limited to user passwords but can also be biomarkers, system patterns or user gestures, could be used for such authentication. On the other hand, since reactive callbacks need to be registered at the sharing server beforehand with proactive invocation, they can be authenticated in the similar way.

4.10 Conclusions

In this chapter, we present a mobile system framework which efficiently interconnects heterogeneous mobile devices towards a personal mobile cloud and supports cooperative resource sharing among these devices. Our basic idea is to allow a mobile application to access remote system resources at another mobile device through remote invocation of existing OS services. Based on the implementation and evaluation over Android OS, we demonstrate that our framework can efficiently support generic resource access between remote mobile devices without incurring any significant system overhead.

Chapter 5

Conclusions

This dissertation explores the methodologies to enhance the capacity of mobile devices through resource sharing in a generic and efficient way. More specifically, mobile devices have limited resource capacity due to the manufacture obstacles, which inhibits the high performance and cannot satisfy the increasing complexity of user applications. Mobile cloud computing is proposed to overcome such limitation in this dissertation that utilizes the remote shared resources to complement and enhance the mobile capacity. On one hand, being integrated into the mobile OS, the proposed frameworks in the dissertation exploit the OS layered architecture to isolate the heterogeneity in mobile hardware and software, which enables generic access to remote resources. On the other hand, the proposed frameworks adapt to the runtime execution patterns to minimize the network transmission of resource data, which hence ensures the efficiency of remote resource access.

In detail, the dissertation presents the solution as three specialized frameworks which tackle the resource sharing of CPU, GPU, and system peripherals respectively.

- **A mobile offloading framework to access CPU on remote cloud:** This novel framework exploits the cloud CPU to accelerate the program executions of mobile applications by offloading computation-intensive methods in Dalvik VM with least context migration. The framework first runs offline parsing to application binaries so as to identify the relevant memory contexts for a specific method. Such parsing results are loaded at runtime to screen the thread stack and heap contexts, which enables the framework to migrate only such relevant memory contexts to the remote cloud. The proposed framework is implemented over practical Android OS, and the experimental results over realistic smartphone applications show that the system reduces 70% of memory context migration than existing schemes, without compromising the offloading effectiveness.

- **A mobile VR framework to access GPU on remote cloud:** *DeltaVR* is a systematic mobile VR framework that maximizes mobile VR performance by utilizing the remote cloud GPU to render high-quality graphics. The framework utilizes the cloud's computational power to explicitly decide the pixel redundancy across adjacent frames and eliminate such redundant pixels during frame transmission, which hence transmits only the distinct portions of each frame to mobile devices. The framework is implemented as a mobile middleware over the Android OS and Unity game engine. The experimental results over real-world VR applications show that DeltaVR reduces more than 95% of the VR frame data transmission, while providing satisfactory VR experience.

- **A framework to access mobile peripherals on personal cloud:** This sharing framework allows the mobile devices owned by a user to be interconnected as a personal mobile cloud so that these devices can complement each other with seamless remote peripheral access. Such interconnection is achieved through the remote invocation of unified system services being defined in mobile OS. The proposed framework is implemented as a middleware on Android OS over various mobile platforms with diverse characteristics and resource limits. The evaluation results show that the design can efficiently support ubiquitous access to peripheral resources between remote systems for arbitrary mobile applications, without incurring any significant system overhead.

Bibliography

[tim] Asynchronous TimeWarp. https://developer.oculus.com/documentation/mobilesdk/latest/concepts/mobile-timewarp-overview/. 58

[fov] FOV of VR headsets. https://virtualrealitytimes.com/2017/03/06/chart-fov-field-of-view-vr-headsets/. 44

[lit] Lite. https://assetstore.unity.com/packages/3d/environments/fantasy/make-your-fantasy-game-lite-8312. 37

[Mir] Miracast. http://www.wi-fi.org/discover-wi-fi/wi-fi-certified-miracast. 85

[pim] Pimax 8K VR. www.pimaxvr.com/8k/. 58

[sci] Sci-Fi Modular Environment. https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-modular-environment-3426. 37

[vik] Viking Village. https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140. 37, 44

[wow] Wowza Streaming Cloud. https://www.wowza.com/solutions/streaming-types/virtual-reality-and-360-degree-streaming/. 53

[x26] x264. http://www.videolan.org/developers/x264.html. 48

[10] Abari, O., Bharadia, D., Duffield, A., and Katabi, D. (2016). Cutting the cord in virtual reality. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 162–168. ACM. 57

[11] Abari, O., Bharadia, D., Duffield, A., and Katabi, D. (2017). Enabling high-quality untethered virtual reality. In *NSDI*, pages 531–544. 57

[12] Amiri Sani, A., Boos, K., Yun, M. H., and Zhong, L. (2014). Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of ACM MobiSys*. 3, 62, 82, 85

[Android Developers] Android Developers. Bytecode for the Dalvik VM. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. 17

[14] Arnau, J.-M., Parcerisa, J.-M., and Xekalakis, P. (2014). Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 529–540. IEEE. 59

[15] Baratto, R. A., Kim, L. N., and Nieh, J. (2005). THINC: a virtual display architecture for thin-client computing. *ACM SIGOPS Operating Systems Review*, 39(5):277–290. 2, 3, 62, 84

[16] Boos, K., Chu, D., and Cuervo, E. (2016). Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 291–304. ACM. 32, 34, 57

[17] Carmack, J. (2013). Latency mitigation strategies. *Twenty Milliseconds*. 59

[18] Chen, K.-T., Huang, P., and Lei, C.-L. (2006). Game traffic analysis: An mmorpg perspective. *Computer Networks*, 50(16):3002–3023. 59

[19] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM. 3, 8, 9, 11, 32

[20] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proceedings of USENIX NSDI*, pages 273–286. 11

[21] Cordeiro, P. J., Gomez-Pulido, J., and Assunção, P. A. (2008). Efficient constrained video coding for low complexity decoding. In *International Conference Image Analysis and Recognition*, pages 243–252. Springer. 39

[22] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010). Maui: making smartphones last longer with code offload. In *Proceedings of ACM MobiSys*. 1, 2, 8, 9, 10, 11, 32, 57, 61

[23] Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., and Musuvathi, M. (2015). Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135. ACM. 1, 34, 44, 57

[24] Dascal, J., Reid, M., IsHak, W. W., Spiegel, B., Recacho, J., Rosen, B., and Danovitch, I. (2017). Virtual reality and medical inpatients: A systematic review of randomized, controlled trials. *Innovations in clinical neuroscience*, 14(1-2):14. 32

[25] Dempsey, P. (2016). The teardown: Htc vive vr headset. *Engineering & Technology*, 11(7-8):80–81. 32

[26] Desai, P. R., Desai, P. N., Ajmera, K. D., and Mehta, K. (2014). A review paper on oculus rift-a virtual reality headset. *arXiv preprint arXiv:1408.1173*. 32

[27] Dhome, M., Richetin, M., Lapreste, J.-T., and Rives, G. (1989). Determination of the attitude of 3d objects from a single perspective view. *IEEE transactions on pattern analysis and machine intelligence*, 11(12):1265–1278. 35

[28] Dinh, H. T., Lee, C., Niyato, D., and Wang, P. (2013). A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611. 57

[29] Edwards, W. K., Newman, M. W., Sedivy, J. Z., and Smith, T. F. (2009). Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Transactions on Computer-Human Interaction*, 16(1):3. 62

[30] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. (2010). Diversity in Smartphone Usage. In *Proceedings of MobiSys*, pages 179–194. ACM. 3

[31] Fernando, N., Loke, S. W., and Rahayu, W. (2013). Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106. 61

[32] Gao, W., Li, Y., Lu, H., Wang, T., and Liu, C. (2014). On exploiting dynamic execution patterns for workload offloading in mobile cloud applications. In *Proceedings of IEEE ICNP*. 8

[33] Giurgiu, I., Riva, O., Juric, D., Krivulev, I., and Alonso, G. (2009). Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Proceedings of the 10th International Middleware Conference*. 8

[34] Gordon, M. S., Hong, D. K., Chen, P. M., Flinn, J., Mahlke, S., and Mao, Z. M. (2015). Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 137–150. ACM. 32

[35] Gordon, M. S., Jamshidi, D. A., Mahlke, S. A., Mao, Z. M., and Chen, X. (2012). Comet: Code offload by migrating execution transparently. In *Proceedings of OSDI*, pages 93–106. 3, 8, 9, 10, 11, 13, 23, 26, 28, 30, 32, 57

[36] Grosse-Puppendahl, T., Herber, S., Wimmer, R., Englert, F., Beck, S., von Wilmsdorff, J., Wichert, R., and Kuijper, A. (2014). Capacitive near-field communication for ubiquitous interaction and perception. In *Proceedings of ACM UbiComp*, pages 231–242. ACM. 84

[37] Grossman, G. E., Leigh, R. J., Abel, L., Lanska, D. J., and Thurston, S. (1988). Frequency and velocity of rotational head perturbations during locomotion. *Experimental brain research*, 70(3):470–476. 45

[38] Ha, K., Chen, Z., Hu, W., Richter, W., Pillai, P., and Satyanarayanan, M. (2014). Towards Wearable Cognitive Assistance. In *Proceedings of ACM MobiSys*, pages 68–81. 11

[39] Hamilton, P. and Wigdor, D. J. (2014). Conductor: enabling and understanding cross-device interaction. In *Proceedings of ACM CHI*. 62

[40] Hao, F., Kodialam, M., Lakshman, T., and Mukherjee, S. (2014). Online allocation of virtual machines in a distributed cloud. In *Proceedings of IEEE INFOCOM*, pages 10–18. IEEE. 8

[41] Hardegger, M., Nguyen-Dinh, L.-V., Calatroni, A., Tröster, G., and Roggen, D. (2014). Enhancing action recognition through simultaneous semantic mapping from body-worn motion sensors. In *Proceedings of ACM ISWC*. 1, 61, 84

[42] Horowitz, M., Joch, A., Kossentini, F., and Hallapuro, A. (2003). H. 264/avc baseline profile decoder complexity analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):704–716. 43

[43] Huang, C.-Y., Hsu, C.-H., Chang, Y.-C., and Chen, K.-T. (2013). Gaminganywhere: an open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*, pages 36–47. ACM. 57

[44] Huang, J., Xu, Q., Tiwana, B., Mao, Z., Zhang, M., and Bahl, P. (2010). Anatomizing Application Performance Differences on Smartphones. In *Proceedings of ACM MobiSys*, pages 165–178. 3

[45] J. Zhang, F. R. and Lin, C. (2014). Delay guaranteed live migration of virtual machines. In *Proceedings of IEEE INFOCOM*. IEEE. 11

[46] Judge, A., Nixon, P., Cahill, V., Tangney, B., and Weber, S. (1998). Overview of distributed shared memory. In *Technical Report, Trinity College Dublin*. Citeseer. 11, 73

[47] Kämäräinen, T., Siekkinen, M., Ylä-Jääski, A., Zhang, W., and Hui, P. (2017). Dissecting the end-to-end latency of interactive mobile video applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 61–66. ACM. 32

[48] Kang, S. B. (1998). Survey of image-based rendering techniques. In *Videometrics VI*, volume 3641, pages 2–17. International Society for Optics and Photonics. 46

[49] Kemp, R., Palmer, N., Kielmann, T., and Bal, H. (2012). Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer. 10

[50] Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of INFOCOM*. 1, 2, 8, 9, 10, 11, 32, 61

[51] Kumar, K. and Lu, Y.-H. (2010). Cloud computing for mobile users: Can offloading computation save energy? *IEEE Computer*, 43(4):51–56. 8

[52] Lai, Z., Hu, Y. C., Cui, Y., Sun, L., and Dai, N. (2017). Furion: Engineering high-quality immersive virtual reality on todays mobile devices. In *Proceedings of the 23rd International Conference on Mobile Computing and Networking (MobiCom). ACM, Snowbird, Utah, USA*. 1, 3, 34, 53, 56, 58

[53] Lee, K., Chu, D., Cuervo, E., Kopf, J., Degtyarev, Y., Grizan, S., Wolman, A., and Flinn, J. (2015). Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM. 34

[54] Lee, K., Chu, D., Cuervo, E., Wolman, A., and Flinn, J. (2014). Delorean: using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys)*. 34

[55] Li, Y. and Gao, W. (2015). Code offload with least context migration in the mobile cloud. In *Proceedings of IEEE INFOCOM*, pages 1876–1884. IEEE. 84

[56] Lin, F. X., Wang, Z., and Zhong, L. (2014). K2: a mobile operating system for heterogeneous coherence domains. *ACM SIGARCH Computer Architecture News*, 42(1):285–300. 3

[57] Litke, A., Skoutas, D., and Varvarigou, T. (2004). Mobile grid computing: Changes and challenges of resource management in a mobile grid environment. In *Proceedings of PAKM*. 84

[58] Lu, H. and Gao, W. (2016a). Scheduling dynamic wireless networks with limited operations. In *Proceedings of IEEE ICNP*, pages 1–10. IEEE. 84

[59] Lu, H. and Gao, W. (2016b). Supporting real-time wireless traffic through a high-throughput side channel. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 311–320. ACM. 84

[60] Lu, J., Sookoor, T., Srinivasan, V., Gao, G., Holben, B., Stankovic, J., Field, E., and Whitehouse, K. (2010). The smart thermostat: using occupancy sensors to save energy in homes. In *Proceedings of the 8th ACM SenSys*, pages 211–224. 2, 62

[61] Maor, E. (2007). *The Pythagorean theorem: a 4,000-year history*. Princeton University Press. 47

[62] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM. 49

[63] Mark, W. R., McMillan, L., and Bishop, G. (1997). Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 7–ff. ACM. 37, 55, 58

[64] Merchant, Z., Goetz, E. T., Cifuentes, L., Keeney-Kennicutt, W., and Davis, T. J. (2014). Effectiveness of virtual reality-based instruction on students' learning outcomes in k-12 and higher education: A meta-analysis. *Computers & Education*, 70:29–40. 32

[65] Nehab, D., Sander, P. V., Lawrence, J., Tatarchuk, N., and Isidoro, J. R. (2007). Accelerating real-time shading with reverse reprojection caching. In *Graphics hardware*, volume 41, pages 61–62. 5, 36, 59

[66] Ohta, T.-i., Maenobu, K., and Sakai, T. (1981). Obtaining surface orientation from texels under perspective projection. In *Proceedings of IJCAI*, volume 81, pages 746–751. 35

[67] Qiu, H., Ahmad, F., Govindan, R., Gruteser, M., Bai, F., and Kar, G. (2017). Augmented vehicular reality: Enabling extended vision for future vehicles. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 67–72. ACM. 32

[68] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. (2011). Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of MobiSys*. 11

[69] Reinert, B., Kopf, J., Ritschel, T., Cuervo, E., Chu, D., and Seidel, H.-P. (2016). Proxy-guided image-based rendering for mobile devices. In *Computer Graphics Forum*, volume 35, pages 353–362. Wiley Online Library. 37, 58

[70] Ryu, U., Ahn, K., Kim, E., Kim, M., Kim, B., Woo, S., and Chang, Y. (2013). Adaptive step detection algorithm for wireless smart step counter. In *Information Science and Applications (ICISA), 2013 International Conference on*, pages 1–4. IEEE. 5

[71] Satyanarayanan, M. (2011). Mobile computing: the next decade. *ACM SIGMOBILE Mobile Computing and Communications Review*, 15(2):2–10. 1, 8, 11

[72] Schaub, F., Könings, B., Lang, P., Wiedersheim, B., Winkler, C., and Weber, M. (2014). PriCal: context-adaptive privacy in ambient calendar displays. In *Proceedings of ACM UbiComp*, pages 499–510. 62

[73] Schmidt, C. and Parashar, M. (2004). A peer-to-peer approach to web service discovery. *World Wide Web*, 7(2):211–229. 71

[74] Shea, R., Liu, J., Ngai, E. C.-H., and Cui, Y. (2013). Cloud gaming: architecture and performance. *IEEE network*, 27(4):16–21. 57

[75] Shi, C., Lakafosis, V., Ammar, M. H., and Zegura, E. W. (2012). Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pages 145–154. ACM. 1, 61

[76] Skupin, R., Sanchez, Y., Hellge, C., and Schierl, T. (2016). Tile based hevc video for head mounted displays. In *Multimedia (ISM), 2016 IEEE International Symposium on*, pages 399–400. IEEE. 58

[77] Sreedhar, K. K., Aminlou, A., Hannuksela, M. M., and Gabbouj, M. (2016). Viewport-adaptive encoding and streaming of 360-degree video for virtual reality applications. In *Multimedia (ISM), 2016 IEEE International Symposium on*, pages 583–586. IEEE. 58

[78] Tong, L., Li, Y., and Gao, W. (2016). A hierarchical edge cloud architecture for mobile computing. In *Proceedings of IEEE INFOCOM*, pages 91–99. IEEE. 84

[79] Vallina-Rodriguez, N. and Crowcroft, J. (2011). Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, pages 37–42. ACM. 84

[80] Vázquez, C., Tam, W. J., and Speranza, F. (2006). Stereoscopic imaging: filling disoccluded areas in depth image-based rendering. In *Proc. SPIE*, volume 6392, page 63920D. 58

[81] Viola, I., Kanitsar, A., and Groller, M. E. (2003). *Hardware-based nonlinear filtering and segmentation using high-level shading languages*. IEEE. 49

[82] Wang, E. J., Lee, T.-J., Mariakakis, A., Goel, M., Gupta, S., and Patel, S. N. (2015). Magnifisense: Inferring device interaction using wrist-worn passive magneto-inductive sensors. In *Proceedings of ACM UbiComp*, pages 15–26. 1, 61

[83] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612. 44, 55

[84] Wiegand, T., Sullivan, G. J., Bjontegaard, G., and Luthra, A. (2003). Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576. 33, 39

[85] Xiang, L., Ye, S., Feng, Y., Li, B., and Li, B. (2014). Ready, set, go: Coalesced offloading from mobile devices to the cloud. In *Proceedings of IEEE INFOCOM*. IEEE. 10

[86] Zheng, Y.-L., Ding, X.-R., Poon, C. C. Y., Lo, B. P. L., Zhang, H., Zhou, X.-L., Yang, G.-Z., Zhao, N., and Zhang, Y.-T. (2014). Unobtrusive sensing and wearable devices for health informatics. *Biomedical Engineering, IEEE Transactions on*, 61(5):1538–1554. 2, 62

[87] Zhong, R., Wang, M., Chen, Z., Liu, L., Liu, Y., Zhang, J., Zhang, L., and Moscibroda, T. (2017). On building a programmable wireless high-quality virtual reality system using commodity hardware. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM. 32, 33

[88] Zhou, H., Fu, Y., and Liu, C. (2015). Supporting dynamic gpu computing result reuse in the cloud. In *HotCloud*. 59

[89] Zhu, F., Mutka, M. W., and Ni, L. M. (2005). Service discovery in pervasive computing environments. *IEEE Pervasive computing*, (4):81–90. 71

Vita

Yong Li received the B.E. degree from East China Normal University in 2008, majoring in software engineering. After then, he worked as a software engineer in industry for almost five years, during which he had rich experiences in the development of enterprise application systems. To embrace bigger challenges and further develop the career, from 2013, he started pursuing his Ph.D. degree in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research focuses on mobile cloud computing, which aims to bridge the mobile systems and cloud architecture with efficient remote resource access.