공학박사 학위논문

Execution Offloading Techniques to Optimize Mobile Cloud Computing

모바일 클라우드 컴퓨팅 최적화를 위한 실행 오프로딩 기법

BY

양 승 준

2015 년 2 월

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

공학박사 학위논문

Execution Offloading Techniques to Optimize Mobile Cloud Computing

모바일 클라우드 컴퓨팅 최적화를 위한 실행 오프로딩 기법

BY

양 승 준

2015 년 2 월

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Execution Offloading Techniques to Optimize Mobile Cloud Computing

모바일 클라우드 컴퓨팅 최적화를 위한 실행 오프로딩 기법

지도교수 백 윤 흥

이 논문을 공학박사 학위논문으로 제출함

2014 년 11 월

서울대학교 대학원

전기 컴퓨터 공학부

양 승 준

양 승 준의 공학박사 학위논문을 인준함

2014 년 12 월

위 원 장	문수묵
부위원장	백윤흥
위 원	윤성로
위 원	정수환
위 원	김장우

Abstract

Smartphones and tablets are rapidly becoming the computing device of preference in the global internet connected device market. Following the trend of the time, the users spend more time on these smart mobile devices (SMDs) using highly sophisticated applications, such as vision, graphics and augmented reality. It is still challenging to deliver such complex applications on SMDs, however, due to the key resource constraint like limited battery and low network bandwidth. In order to tackle this problem, recent studies suggested *mobile cloud computing* techniques that attempt to connect resource-constrained SMDs to nearby resource-rich powerful clouds. These techniques often imply *execution offloading* (or *computation offloading*), which is a promising technique to effectively deliver mobile cloud computing into the real-world mobile computing environments.

The main purpose of execution offloading is to throw the computational burden of SMD to the powerful servers by migrating a process or executing a method remotely. To achieve this goal, the current application state is captured and transferred to the servers over the network at runtime in execution offloading. Expectedly, the state transfer cost for the application state is a deciding factor for the success of execution offloading; because the size of the application state may reach up to multi-megabytes at a time, reducing the transferred state size is very important to maximize the benefit of execution offloading. In this dissertation, I propose novel techniques based on compiler code analysis that effectively reduce the state transfer cost by transferring only the essential application state actually referenced in the servers.

Another observation for execution offloading is that the early offloading studies depend on many idle assumptions. For example, they assume that the performance of a target server is always idle and constant. In the real-world commercial cloud environments, however, the cloud provider tries to maximize the server throughput by running as many applications as possible on a single server (i.e., *oversubscription*) and it makes such assumptions unrealistic. To design more realistic offloading scheme for the real-world cloud environments, therefore, it is necessary to consider the cost-effective behavior of the cloud platform. In this dissertation, I introduce a new cost-effective execution offloading scheme, called *CMcloud*, which not only maximizes the server throughput but also satisfies the post-offload performance of all target applications.

One challenge in execution offloading is to design the application-specific offloading techniques. Many mobile applications have their own, unique characteristics and some of them may make the strategy of the existing studies fail. It is important to adopt target-specific optimizations into offloading framework, therefore, to improve further the performance of target applications via execution offloading. To show the opportunity to achieve this goal, I suggest a streaming-based execution offloading framework that successfully guarantees quality of service (QoS) of 3D video games. I further propose *live offloading*, which allows transferring the current application state before the remote execution of the offloaded application actually begins, to make the suggested framework even more effective for better user experience.

Keywords: mobile cloud computing, execution offloading, code analysis, cloud environment, application-specific optimization, 3D video game
Student Number: 2008-20913

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Execution Offloading for Mobile Cloud Computing

Mobile applications are a steadily growing segment of the global software market, whose revenue is expected to reach more than 25 billion dollars by 2015 [1]. As their processing capabilities increase, smartphones and tablets are rapidly becoming the computing device of preference that can accommodate most up-to-date mobile applications. Even so, it is still challenging to deliver highly sophisticated applications these smart mobile devices (SMDs) due to the key resource constraints like limited battery, poor processing power and low network bandwidth.

To alleviate this problem, latest studies suggested various *mobile cloud computing* techniques that attempt to connect resource-constrained mobile devices to nearby resource-rich powerful clouds [2, 3, 4, 5]. The basic idea is to let devices leverage computation and energy on cloud servers to execute (part of)

mobile code that requires heavy use of computing or network resources. People believe that mobile cloud computing opens a new world where SMDs armed with various network connections and rich sensors will extend dramatically its functionalities with help of computational power of clouds. First of all, it seems obvious that mobile cloud computing can accommodate a much wider range of complex applications which have been impractical to run solely on SMDs, such as perception applications, vision, graphics, healthcare, augmented reality and m-learning [5, 6]. As another advantage, mobile cloud computing may relax the design constraints of smartphone hardware which, due to the considerations of size, costs and battery capacity, have been strictly imposed on hardware features like CPU, storage and network [7].

To execute mobile code on the remote server such as the cloud or wall-powered PCs, previous work has often employed a technique, called *execution offloading*, which is the act of transferring execution (or process) between two machines during its run time. By relieving computational loads, the technique labors to bring SMDs benefits in terms of battery and execution time from the servers in their proximity. In execution offloading, there are two key tasks involved before remote execution: *code partitioning* and *state migration*. In recent years, there has been a great deal of research conducted to find or support optimal partitioning of distributed systems with mobile devices. Some researchers [8] proposed *static* partitioning approaches where the job assigned to each machine in the system is fixed at compile time. Static partitioning ought to be more doable if the computational resource configurations such as processor speed, memory capacity, energy consumption and network characteristics, remain fairly constant once the process is launched. In mobile computing, however, the configurations can be changed due to user mobility even in the middle of process execution. Therefore, mostly other works [2, 3, 6, 9] have been on the

development of *dynamic* or semi-dynamic partitioning approaches for execution offloading.

1.2 Techniques to Minimize State Transfer Costs

In dynamic execution offloading approach, which code regions (e.g., methods or functions) actually run on the server is decided at run time when the resource configurations for execution become known. Once a certain region is finally selected at run time, the current application state for execution needs to be captured and migrated to the server along with the control command that directs the resumption of the execution. In one approach [3], the entire state including the existing stack and all *reachable* heap objects is migrated to offload the full process. In the other one [2], the stack is not to be migrated as the functions set to run remotely will be newly invoked in the server. Clearly, there are trade-offs between these two approaches. Above all, the usual amount of state transferred, which is a major decisive factor for the efficacy of execution offloading over mobile networks, is smaller in the latter. In contrast, the former approach relies little on users for code alteration, and supports more versatile code execution because, with the full process in its hand, the server would be able to control the execution more adaptively.

In ideal cases where the costs for transferring the application state can be neglected, any code regions except for those using the device-only resources like GPS and screens would benefit from remote execution. This is obvious because the server processor speed is much faster, and virtually no energy of the SMD would be consumed while they run on the server. In reality, however, the state transfer costs may not be neglected but even be a dominant factor that inhibits the regions from executing remotely. Expectedly, the state transfer

costs are roughly proportional to the size of the application state; because the size of the application state may reach up to multi-megabytes at a time in some approaches [3], reducing the transferred state size is very important to maximize the benefit of execution offloading.

In this dissertation, I propose novel techniques based on compiler code analysis that effectively reduce the state transfer cost, and so help us offload more code regions for lowering the total execution time or energy consumption. This has been achieved by only transferring the *essential* heap objects, which are defined to be the reachable objects that will be possibly accessed within the remotely executing code regions. Moreover, I introduce *partial stack transfer*, which is a technique for reducing the costs even more by transmitting only the frames actually referenced in the cloud, rather than transferring the entire stack. The experimental results seem promising; the state transfer costs are reduced on average by a factor of ten. As a result, our mobile code was able to be offloaded more aggressively at run time, attaining an overall speedup of process execution up to seven.

1.3 Cloud Platform for Cost-Effective Execution Offloading

In order to focus on building the basic concepts of execution offloading, the early offloading studies depend on many idle assumptions. For example, they assume that a target server is always available for free of charge, the server's load is always idle or stable, the application has been previously profiled for the target server, and the post-offload performance matches the user-expected performance.

However, such assumptions are unrealistic for real-world commercial cloud

environments, where the cloud provider charges the users based on their cloud resource usage and each server aims to run as many applications as possible to maximize the server throughput or minimize the server costs (i.e., *oversubscription*). If a target server is running multiple applications, the pre-profiled performance of the offloaded application will not match the actual post-offload performance, which leads to a critical Quality-of-Service (QoS) failure. On the other hand, if the cloud provider forces to maintain the initial profiling state of servers (e.g., idle or static load) it fails to increase the server throughput, which leads to the increased server costs and the user service fee.

The execution offloading approaches for the real-world commercial cloud, therefore, have to maximize the server throughput and satisfy the post-offload performance of all target applications at the same time. To achieve this goal, I introduce *CMcloud*, a novel cost-effective execution offloading platform. The key idea of CMcloud is to exploit a novel performance modeling methodology for accurately estimating the post-offload performance of the target application on any target server, regardless of its current utilization. Simultaneously, CMcloud allows to offload as many applications to each server as possible without violating the target applications' pre-profiled performance. If the target performance cannot be achieved using the currently allocated server due to inaccurate performance estimations, CMcloud performs fast inter-server live migrations to achieve the target performance. In this way, CMcloud can offer to users its QoS-guaranteed offload service at a very low price, while minimizing the cloud operation costs.

1.4 Application-Specific Execution Offloading

The existing studies have shown that execution offloading can accommodate a much wider range of complex applications, by improving the runtime performance of these applications on SMDs with the powerful cloud or servers. There are still many challenges in execution offloading, however, to improve further the performance of mobile applications. One of those challenges is the application-specific execution offloading. Because many mobile applications usually have their own, unique characteristics which are closely related to their runtime performance, it is important to adopt those characteristics into offloading scheme to fully exploit the benefit of execution offloading.

To show the opportunity to achieve this goal, I propose a streaming-based execution offloading framework that successfully guarantees quality of service (QoS) of 3D video games. There are two reasons why I chose 3D video games as the target application. First, 3D video games are one of the popular applications extended to SMDs; mobile games are the fastest-growing segment of the video game market, with revenue set to nearly double between 2013 and 2015 from $13.2 billion to $22 billion dollars [10]. Second, 3D video games have a noticeable characteristic which distinguishes them from other applications. Their major and time-consuming functions, called *rendering*, continue to generate a lot of images while the game is running and as a result, the state transfer cost of them is quite huge. Such a high cost may make the existing offloading schemes avoid to offloading those rendering functions, even though they take the majority of the overall execution time of 3D video games and also requires powerful graphics processing unit unit (GPU).

Based on streaming techniques, the offloading framework I propose enables execution offloading for 3D video games by reducing the state transfer cost of

rendering functions. When the rendering functions are being offloaded continuously, the generated images are streamed to the SMD and only the newly-update application states like the user inputs are transferred to the server. As a result, the proposed framework effectively offloads rendering functions and successfully guarantees QoS of 3D video games in terms of execution time. I further introduce *live offloading*, which allows transferring the current application state before the remote execution actually begins, to make the proposed framework even more effective for better user experience. The manipulated application state during the remote execution is also returned before the remote execution is finished. With live offloading, the large data transfer cost at the beginning and end of remote execution can be hidden; it prevents that such a large data cost degrades user experience by enlarging response time.

Chapter 2

Techniques to Minimize State Transfer Costs for Execution Offloading

As mentioned in the previous chapter, the current application state should be transferred for remote execution and the cost for transferring the state is closely related to the efficiency of execution offloading. Therefore, it is important to reduce the state transfer cost for more effective execution offloading. The key idea to achieve this goal is transferring only the essential objects (e.g., heap objects or stack frames), which I define to be the objects that will be possibly accessed within the remotely executing code regions.

The rest of this chapter is organized as follow: first, I explain the baseline offloading approach, CloneCloud [3], and how adversely the size of the application state affects the performance of execution offloading. In the following subsection, I propose novel techniques that reduce the amount of transferred state by identifying essential objects from the mobile code based on compiler code anal-

ysis. I also introduce a technique called partial stack transfer, and explain how this technique reduces the state transfer cost even more by transmitting only the frames actually referenced in the cloud, rather than transferring the entire stack. The experimental results, discussions, and related work is also presented.

2.1 Background: Reachable State Transfer

In this section, we discuss how adversely the size of migrated state affects the performance of execution offloading.

2.1.1 Overview of CloneCloud

In CloneCloud, a process is an Android phone application running on the Dalvik virtual machine (VM). The process may comprise multiple threads, and some of them, which we call *migratable threads*, contain the remotely executable methods (REMs) in their code. If a thread has no REM, it will be herein called a *resident thread*. As a rule, CloneCloud declares a method to be an REM if it does not need to access local resources in the phone such as GPS, cameras and screen. However, the decision on what methods actually will run remotely on the cloud sever is deferred until the process starts execution when its computational resources are all revealed. For this decision, CloneCloud implemented two components: profiler and solver.

The profiler measures expected execution times and energy usages of all REMs on the mobile device as well as on the server. It also calculates the average cost required to offload a migratable thread; this cost is not only changed depending on the REM but also influenced by network characteristics, such as loss rate, latency and bandwidth. The solver accepts all the performance numbers obtained during profiling, and outputs optimal *partitions* for each thread.

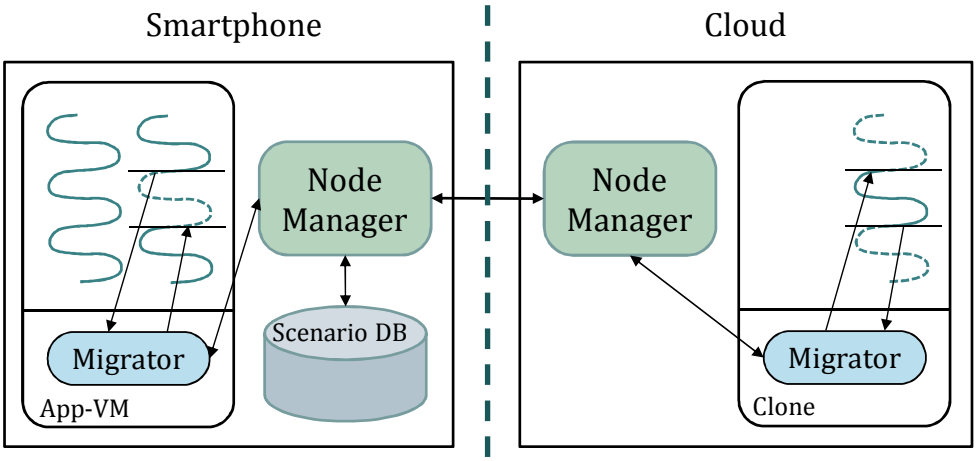


Figure 2.1 Migration overview of CloneCloud

A partition is an execution scenario consisting of a sequence of decisions made at every REM on whether the REM should be offloaded or not. Among many candidate partitions, the solver chooses the best one that minimizes the overall run time and/or energy consumption of the mobile device. Certainly, the best partitions for the same thread may vary according to the network interfaces which the mobile devices are connected. Therefore, the solver generates a collection of partitions each optimized for a unique network condition, like 3G or Wi-Fi.

The resulting optimal partitions are stored into the scenario database (DB) shown in Figure 2.1. When a process is launched, CloneCloud detects the current network status and retrieves the DB to fetch the optimal partition for this process execution. During the execution, if a thread is to be migrated according to the scenario extracted from the DB, the migrator suspends the migrating thread and captures its state by visiting every stack frame in the existing stack. The state is serialized to generate a *state package* that is then given to the node manager whose mission is to transfer the state package over the network

between machines. The state arriving at the cloud is de-serialized and restored into the memory by the migrator on the server. Associated with the transferred state, a new thread is cloned on the cloud and takes over the execution control from the original thread until it must return the control back to the mobile device according to the scenario. More details about execution offloading in CloneCloud are referred in their literature [3].

2.1.2 Impact of State Package Size on Performance

The state being transferred when a thread is migrated back and forth between machines includes stack, registers, and reachable heap objects (RHOs). Heap objects are composed of class and data objects. A class object is a template describing the behaviors and states that are shared by the objects of its type, and a data object is an instance of a class that contains its local states and methods. RHOs are any heap objects that are accessible or visible in any potential continuing computation. Figure 2.2 presents an example of heap objects along with two stacks for a face detection process that has two threads: `UI` and `Worker`. When the migrator captures the state of a migrating thread, it identifies RHOs by recursively chasing the reference links starting from local data objects in every stack frame of the thread. This procedure is similar to garbage collection. But the migrator looks for live (or reachable) objects, while the garbage collector finds dead ones that have no references to themselves.

In the example, `Worker` currently has two frames, each of which stores local data objects used by one method in the thread at run time. For instance, the frames for two methods `FdView.run` and `FaceDetector.FaceDetection` point to the `FdView` data object and the `FaceDetector` class object, respectively. Each heap object has reference links to its relevant objects. For example, the `FdActivity` data object points to its class object and the `widget/Button` class

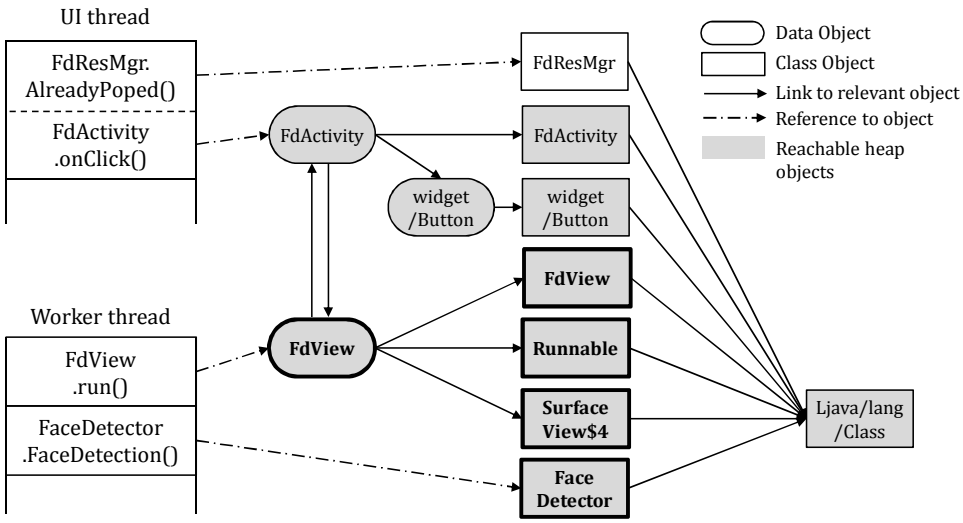


Figure 2.2 An example of Reachable Heap Objects

Table 2.1 Ratio of RHO to total state package

Benchmark	RHOs (KB)	Total state package (KB)	Ratio(%)
Fibonacci	10,545	10,871	97.00
Face detector	10,546	10,870	97.01
Chess engine	10,541	10,855	97.10
FBReader	17,445	17,960	97.13

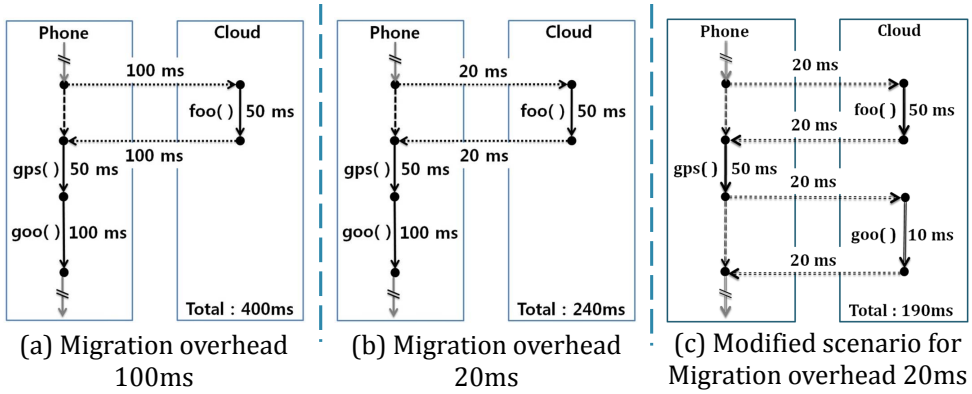
object. With this data structure, RHOs of each method are determined by checking accessibility of heap objects from its stack frame. Note that almost all heap objects in this example are RHOs because the `Fdview` data object has multiple links to other class objects as well as the `FdActivity` data object, which lead to virtually all the other objects in the figure. In Figure 2.2, the RHOs are denoted by shaded boxes. Rectangles stand for class objects while rounded ones for data objects. These RHOs will be captured by the migrator for state transfer if `Worker` is decided to be migrated.

We have discovered that RHOs generally occupy the largest fraction of the state captured by the migrator for transfer. This implies that the node manager would spend most of its time transferring RHOs across the network. Table 2.1 shows the average size of RHOs compared to the entire state package. It confirms our expectation that RHOs take the majority of the state package. Note that such a huge size and high occupancy of RHO is not surprising; an Android application uses at least a few megabytes of heap objects, even though the application is a simple "Hello World" activity [11]. This result suggests that minimizing the size of RHOs should reduce the total state transfer time substantially, thereby contributing the reduction of overall execution time of the migratable thread as well.

Table 2.2 Execution time of each method in the execution scenarios

Methods	Phone	Cloud
foo()	500 ms	50 ms
gps()	50 ms	N/A
goo()	100 ms	10 ms

To explain in more detail the impact of RHO size reduction on the execution time, see the execution scenarios in Figure 2.3, demonstrating that different state transfer times can affect the solver's decision on optimal partitions. We assume here that two methods `foo` and `goo` are REMs while `gps` cannot as it relies on the GPS service on the phone. In Table 2.2, we list the expected execution time of each method estimated by the profiler. In Figure 2.3(a), we list the different optimal partitions that might be produced by the solver depending on the amount of state transfer overhead. If the transfer time is 100 ms, the scenario will be chosen as the optimal partition that minimizes the total execution time. Suppose that the time is cut to merely 20 ms by minimizing the total size of RHOs. Even under the same scenario as in Figure 2.3(a), we can reduce the execution time as depicted in Figure 2.3(b). In reality, however, the solver would choose the one in Figure 2.3(c) as the optimal partition since it further accelerates the performance by dispatching the REM goo into the cloud. From this example, we can see that the reduced RHO size will help the solver to find a better partition that exploits more aggressively the computing resources in the cloud, which ultimately can result in dramatic performance improvement in mobile cloud computing.



(a) Migration overhead 100ms

(b) Migration overhead 20ms

(c) Modified scenario for Migration overhead 20ms

Figure 2.3 Impact of state transfer costs on the decision of partitions

Table 2.3 Size comparison of RHO to EHO

Benchmark	RHOs (KB)	EHOs(KB)
Fibonacci (n=42)	10,545	90
Face detector (99 images)	10,546	91
Chess engine (25 pieces)	10,544	12
FBReader (150,700 words)	17,445	2,130

2.2 Essential State Transfer

In this section, we discuss our novel techniques that help us to drastically reduce the size of state transferred at migration points. The central idea behind them is gleaned from the fact that although RHOs are accessible to a thread, not all of them are actually used at run time, and thus that from the transferred state package any object can be removed which has no chance of being accessed during remote execution on the cloud. For this we define a heap object for a thread, called an *essential heap object* (EHO), to be a RHO that has explicit references in the thread code. Table 2.3 compares the size of a RHO set and that of an EHO set for the same thread. From the results in the table, we gleaned the fact that EHOs can be much less in number than RHOs in some applications, as will be exhibited in our experiments where we significantly reduce the state transfer time by not transferring all RHOs, but instead transferring only EHOs.

In the subsections below, we first describe a code analysis technique that is used to extract EHOs from RHOs, and then other techniques that enable us to additionally minimize the time to transfer EHOs.

2.2.1 Essential Heap Objects

Among RHOs, many class objects come from super classes because Java class objects usually inherit various states and behaviors from their super classes. But in most cases, all the objects defined in the super classes are not required to execute a process; that is, there are some variables or methods never accessed throughout the whole execution. Hence, the cloned thread will safely run on the cloud even if we do not transfer any RHO that will not be referenced by the thread. However, it is almost impossible for us to statically identify which RHO is to be actually referenced at run time. Therefore in our work, we only remove from the transferred state package every super class object (and its related data objects) that has no reference in any method of a migrating thread. By definition above, all the RHOs remaining in the package automatically become the EHOs. According to our experiments, even this conservative approach to isolate EHOs has reduced the state package size to a large extent.

In our work, the unreferenced objects are simply determined by code analysis, where the names of class objects referenced in every method are all stored into a table. When the migrator captures the state, it searches for RHOs by chasing down the relation links in their class hierarchy. When it comes across a class object, it looks up the object in the table. If the object is not found, it is classified as unreferenced. To identify EHOs by finding unreferenced objects, we propose in this work a new component, called the *state transfer optimizer*, that can be added to the original CloneCloud. Figure 2.4 shows a new system augmented with the optimizer.

In Figure 2.2, we showed an example of RHOs for the `Worker` thread. Here, we also represent the EHOs selected from them with the rectangles or rounded rectangles enclosed by thick lines. Notice that the `FdActivity` data object and

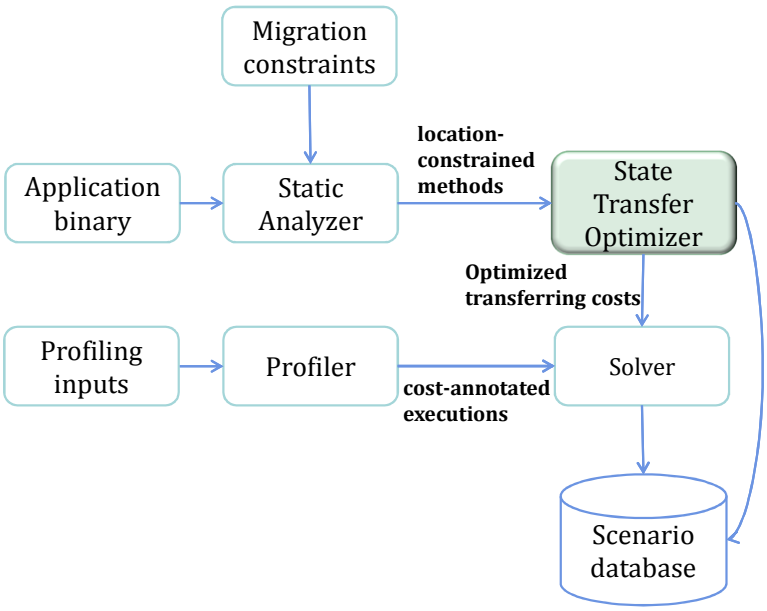


Figure 2.4 State transfer optimizer for execution offloading

its relevant class objects are not chosen as EHOs. This is because the state transfer optimizer reveals through code analysis that a method `Fdview.run` never references them during its execution.

2.2.2 Liveness of Essential Heap Objects

Basically EHOs must be all serialized into a state package because they are literally essential to computation. But a data flow analysis advises us the possibility of further optimization of state transfer by excluding some EHOs from the package. To explain this, we introduce a notion of *liveness* for heap objects. Variables declared local to a method are the local data objects that get stored in a stack frame when the method is invoked. As the case of `Fdview` in Figure 2.2, we have collected RHOs by following the relation links starting from these local data objects. A local data object (or variable) is here said to be *live*

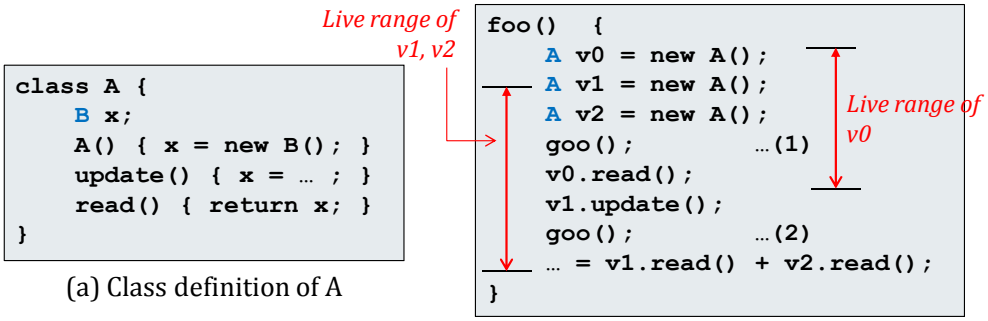


Figure 2.5 Java code for an REM goo

at a point if there are uses of that object after the point in the code. Otherwise it will be considered *dead*. When we assemble a state package with EHOs, we only include the live ones obviously because the dead objects will never be used during the rest of execution of a migrated thread even if they are transferred to the cloud.

Depending on migration points in a method, a data object might be live or dead. For instance in Figure 4.3(b), three data objects local to a method `foo` are listed, and the ranges of their liveness are also pictured. If `goo` is an REM, two invocations to `goo` inside `foo` would be the migration points. Suppose that their thread is to migrate at the first invocation site following the execution scenario. Then, they are packaged as EHOs and sent to the cloud since they are all live. However, if migration occurs at the next invocation site, only `v1` and `v2` are live then (see Figure 2.6(a)). As a result, the dead object `v0` is removed from the original state package as shown in Figure 2.6(b) and (c). To compute the live range of a data object, we have applied conventional compiler techniques based on *def-use* analysis [12].

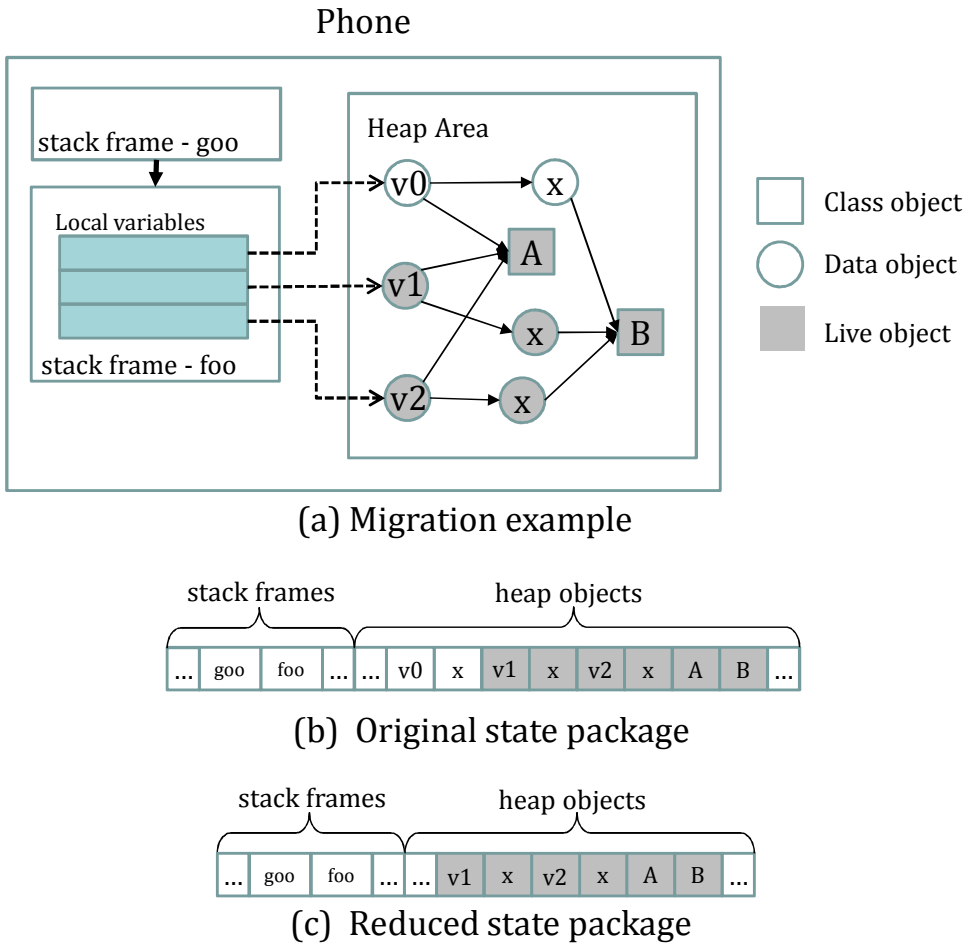


Figure 2.6 Essential heap objects for the Java code

When a local data object is found dead at a migration point, all EHOs related to this object must be examined to determine their liveness (or eligibility of being in the state package). If any of them is also found dead, it will be excluded from the package. As an example, see a data object `x` of class type `B` declared inside class `A` in Figure 4.3(a). If `v0` is not needed in the code, neither is `x` because `x` is exclusively accessed within `v0`. This means that if `v0` is dead, so is `x`. Consequently, as shown in Figure 2.6, `x` was deleted from the package along with `v0`.

2.2.3 Dirty/Clean Essential Heap Objects

Once dead EHOs are all filtered out of the state package, the remaining live ones are finally ready to ship. In this last step of state transfer, we have found a way to save the time and energy of transmitting the package over the network. In order to take a glimpse of this idea, see the code in Figure 4.3(b). Again, let us assume that the thread is about to migrate just before the second invocation of `goo`. In the code, we can see that `v1` has been modified before the migration point while `v2` is still intact. The idea here is that it is not necessary to copy and deliver `v2` from the phone to the cloud because the exactly same content of `v2` can be duplicated simply by creating `v2` on the remote site. We call this unmodified object *clean* and the modified one *dirty*. In our work, we have used a well-known compiler *side-effect* analysis technique [12] to identify which objects have been modified before reaching each migration point.

The analyzer accepts application bytecode as its input to identify dirty and clean objects. First, the analyzer seeks every method call (or, a migration point) in a method when it explores its input code. For each local data object in a method, it labels the object and its relevant objects as dirty is there is any instruction which assigns any value to the object before corresponding

migration point. After every dirty object in a method is identified, the analyzer stores the ID of each dirty object into a table with the address of corresponding migration point. For instance in Figure 4.3(b), the ID of the dirty object `v2` and the address of the second invocation of `goo` are stored to the table. Notice that relevant heap objects to `v2` are not stored.

As stated in Section 2.2.2, live EHOs are only included to the state package in our work. When the migrator chases every stack frame and packs each live EHO in a frame to the package, it also confirms whether the object is dirty or not by searching for the object in the table. If the ID of the object is not in the table, then the object is clean. In this case, the migrator creates a *stub* for the clean EHO and adds it to the state package, instead of adding the EHO itself and its relevant objects. A stub contains information of an object such as ID, class name, and the address of an object necessary for the migrator on the cloud to create new instance of clean EHO. Comparing to the size of clean EHO and its relevant objects, the size of a stub is much smaller, being only a few bytes. For the reason, the state package size can be reduced by substituting the stubs for clean EHOs and their relevant objects. From stubs in the reduced state package, the migrator on the cloud side creates new instances of the clean EHOs and links them to the transferred dirty EHOs. We named this on-the-fly instantiation of the clean EHOs *on-site-duplication*.

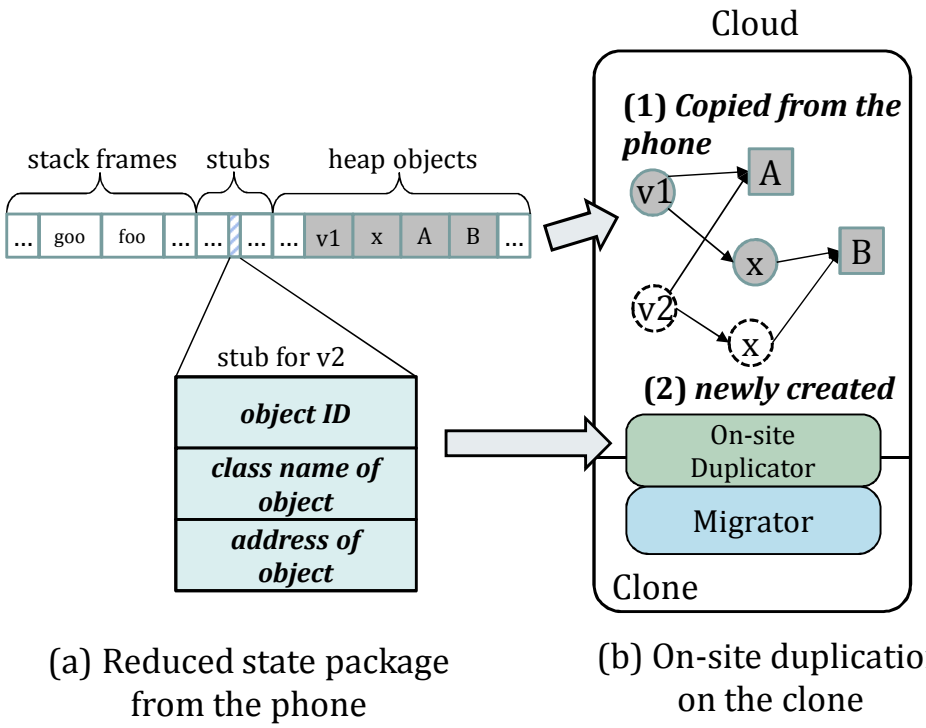


Figure 2.7 State restoration via state copy and on-site duplication

Figure 2.7 shows state restoration via state copy and our on-site duplication, after the migration occurs at the second invocation of `goo` in Figure 4.3(b). Because `v2` is clean in this case, the reduced state package in Figure 2.7(a) does not include `v2` and its relevant object `x`: instead, the stub for `v2` is included in the package. The on-site duplicator in Figure 2.7(b) uses this stub to call the constructor method of `v2`. After this on-site duplication, new instances of `v2` and `x` are created. Then, the migrator restores the complete state for remote execution by assembling them together with the dirty objects `v1` and its relevant objects which are just copied from the phone through the state package.

2.3 Partial Stack Transfer

In the previous section, we explained the definition of EHO and how we reduce the size of transferred state by applying it to execution offloading. We also described the compiler techniques to exclude dead or clean EHOs from the state package to offload more efficiently. In this section, we propose our novel techniques that enable us to partially transfer the existing stack to reduce the state package even more, instead of transmitting the full stack.

2.3.1 Motivation

Although the techniques we mentioned earlier are quite effective in reducing the size of the state package, we still have a chance to make our offloading model more effective. Such a chance is gleaned from the fact that our baseline offloading model, CloneCloud, transfers every stack frame in the existing stack when the migration occurs.

In practice, transferring the entire stack to the cloud is not necessary, because it is rare that every stack frame is referenced in remote execution. The stack frames in the existing stack have the information of the methods which are already invoked but not finished yet. We will call these pre-invoked methods as *method before migration* (MBM). Each frame is transferred to the cloud in case its relevant MBM is finished in remote execution. In most cases, however, only a few MBMs are actually finished in remote site. This is because most mobile applications have a time limit to answer the user's input, so the offloading scenarios for them usually offload only a part of the execution. For this reason, only a part of MBMs is finished and only the relevant stack frames are referenced in remote execution.

Table 2.4 Comparison referenced frames to total frames

Benchmark	The total frames(#)	The referenced frames(#)
SciMark2.FFT	4	1
SciMark2.LU	3	1
SciMark2.SOR	3	1

Table 2.4 compares the number of the captured frames to that of the frames actually referenced in remote execution, according to the optimal offloading scenarios of three benchmarks. As you see, only 25% to 33% of frames are referenced in remote execution. In other words, only 25% to 33% of MBMs are finished in remote. In short, it is usually not necessary to transfer the entire existing stack, because there is little possibility that every MBM is finished in remote execution. If these unnecessary frames are excluded from the state package, their relevant heap objects are consequently removed[1].

Then the question arises, if it is not necessary, why does our baseline offloading model transfer the entire existing stack? It is because that if we remove any stack frame from the state package carelessly, we may cause *semantic inconsistency*; we define semantic inconsistency as the case when the result of remote execution is different to the result of mobile-only execution. To explain this, see Figure 2.8. In the figure, there are three functions: `foo`, `goo`, and `hoo`. The stack frames and relevant heap objects of each method are represented. Assume that the offloading scenario decides to start the remote execution from the entry of `hoo` to the exit of `goo`. Then, `foo` and `goo` are MBMs, and `hoo` is a method which will be called *method after migration* (MAM). Because the

[1]Note that our migrator is very similar to a garbage collector; if the migrator does not capture any stack frames, then all heap objects relevant to the frame are automatically not captured.

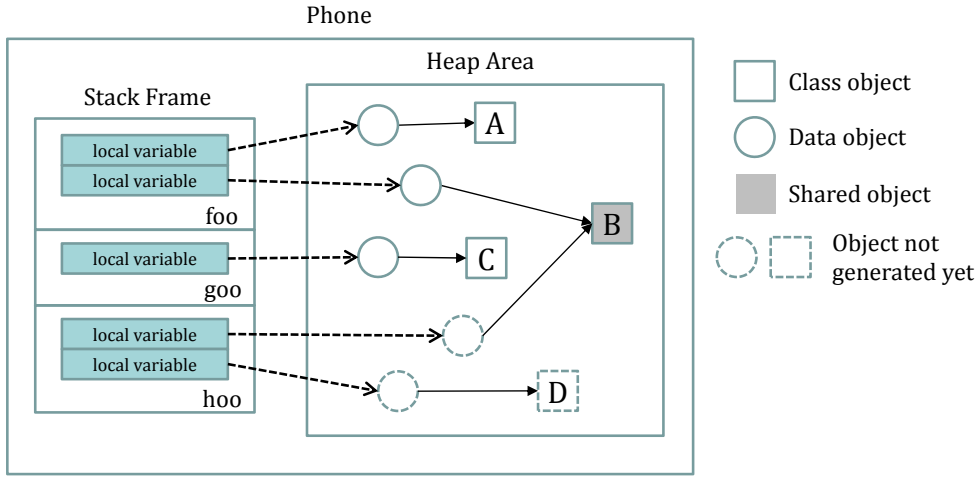


Figure 2.8 An example of semantic inconsistency

remote execution ends at the exit of `goo`, `foo` is never executed and its stack frame is never referenced in cloud as well. At first glance, it seems that simply removing this redundant frame and its relevant objects from the state package does not make any problems in usual cases. In Figure 2.8, however, such a careless optimization may cause semantic inconsistency because `hoo` references one of the relevant class objects of `foo`, `B`, which is denoted in gray. If we exclude `B` from the state package even though `foo` updated the value of it, then `hoo` may get unexpected results due to the out-of-date value of `B`.

In practice, semantic inconsistency can occur when MAMs share static fields of the class objects of MBMs. Unlike a normal field, a static field is not related to a particular instance at all. In other words, a static field has a unique value in a single execution, and the value is shared by all methods referencing the static field. If any method updates the value of a static field, then all other methods are affected. Therefore, if an MBM and an MAM share the same static field and the MBM updates the value of that field, then the field should be included in the state package to let the MAM generate a proper result based on the latest value of the field.

To transfer the existing stack partially, as you see, it is necessary to avoid semantic inconsistency. In the following subsection, we describe our techniques to tackle this semantic inconsistency and how we integrate the techniques into our offloading model so it could support partial stack transfer.

2.3.2 Analysis for Partial Stack Transfer

To avoid semantic inconsistency for partial stack transfer, we must be able to address the following questions:

- Which methods are MAMs?
- Which class objects are referenced by the MAMs?
- How can we effectively adopt the information from these questions to our offloading model?

For the first question, we build a static call graph [12], which is a call graph intended to represent every possible execution flow of the application. It is impossible to exactly predict which methods will be invoked in remote execution, because the behavior of an application can be different in each individual execution. Therefore, we use the static call graph to conservatively figure out every method that can be possibly called in remote execution. If the migration occurs at the entry of a method, then we regard the method and all of its child methods in the static call graph as MAM. Such an approach may not be efficient but is reliable because it guarantees to cover every possible execution flow in the cloud.

The next goal is finding the class objects which are referenced by MAMs. To achieve this goal, we first gather the name of class objects referenced in every method by using the same code analysis described in Section 2.2.1. After that, we build *exclusive reference sets* (ERSs); these sets contain the names of class objects we gather for each method. We also match each ERS to the corresponding nodes in the static call graph. Figure 2.9 shows the static call graph and its ERSs, which are generated for the example in Figure 2.8.

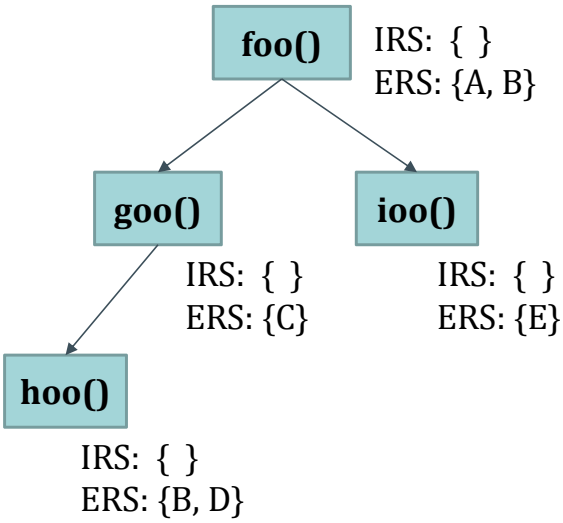


Figure 2.9 Static call graph with ERSs

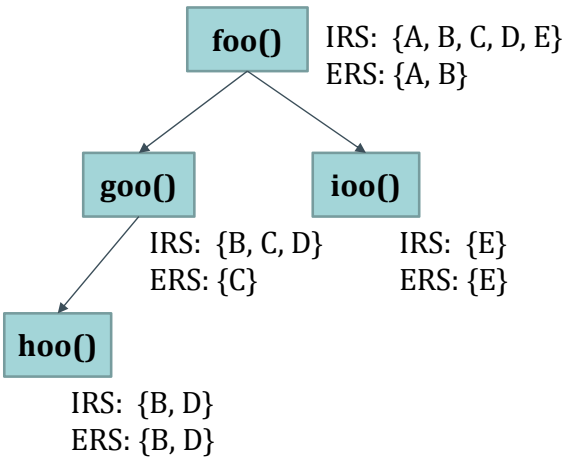


Figure 2.10 Static call graph with ERSs and IRSs

After matching ERSs to their corresponding node is finished, we do depth-first-search (DFS) in post order on the graph to generate *inclusive reference sets* (IRSs) for each node. We define an IRS of a method (or node) as a set of the names of class objects referenced by the method inclusively. In other words, the IRS of a method contains the names of every class object which may be possibly referenced after the method is called in remote execution. In Figure 2.10, for example, the IRS of `goo` includes `B`, `C`, and `D`, which are referenced by `goo` itself (`C`) and its child method `hoo` (`B`, `D`). So, for the node we visit during the DFS, we sum up every ERS of its child nodes to generate the IRS of it. Notice that we generate the IRSs statically; it is because to hasten the run time prediction of which class objects will be referenced by MAMs. If we compute the IRS by gathering the ERSs at run time, the performance of execution offloading may be degraded. The final static call graph and its ERSs and IRSs are represented in Figure 2.10.

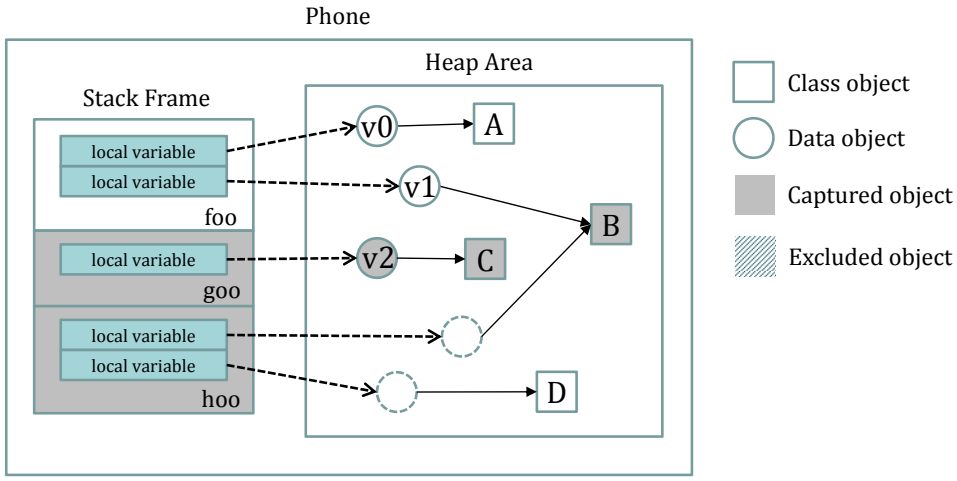
Now, we move on to the last problem which is how we adopt the static call graph and its ERSs and IRSs to our offloading model. First, the migrator goes through the existing stack to figure out which MBMs were already called when the migration occurs. Notice that we do not capture any heap objects here; we just capture the names of MBMs in chronological order. After that, we search the static call graph to find the path that matches the order of the MBMs we got. For example, we have the path from `foo` to `hoo` in Figure 2.10 for the stack in Figure 2.8. Through these steps, we can pick the IRS of the first MAM, which is the first executed method in remote execution. So, the IRS we pick represents all of the class objects that could be possibly referenced in that particular remote execution. In Figure 2.8, `hoo` is the first MAM.

After finding the path and the IRS of the first MAM, the migrator traverses the existing stack once more from top to bottom, to capture the state. For each

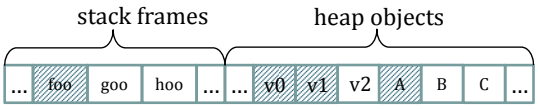
frame it meets during the capture process, the migrator identifies whether the MBM of the frame will be finished in the cloud or in the mobile, based on the offloading scenario. For the former, the migrator captures the frames and their EHOs in the same way mentioned in Section 2.2.1. For the latter, the migrator packs up their relevant class objects if and only if the class objects are the elements of the IRS of the first MAM.

Figure 2.11 shows the result of partial stack transfer applied to the example in Figure 2.8. In the figure, the migrator starts its state capture process from `hoo`. For `hoo`, the migrator keeps the IRS of `hoo` because `hoo` is the first MAM. For `goo`, the migrator captures its frame and relevant objects including `C`, because `goo` is finished in the cloud. Finally, the migrator meets `foo`, which is finished in the mobile, so it chases down the class objects of `foo` and checks the class objects included in the IRS. As a result, the frame of `foo` and all of its objects except `B` are removed from the state package.

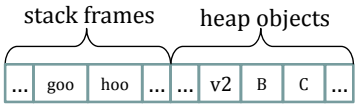
Based on the techniques proposed in this section, we successfully avoid semantic inconsistency and transfer the existing stack partially. In the following section, we discuss the impact of the partial stack transfer on the size of the state package.



(a) Patial stack transfer example



(b) The original state package of full stack transfer



(c) The reduced state package of partial stack transfer

Figure 2.11 Partial stack transfer example

2.4 Evaluation

We implemented our execution offloading model including the state transfer optimizer and the modified migrator on the Android 4.0.3 branch, and tested it on the smartphone and the server, respectively. The smartphone is a Galaxy Nexus with dual-core 1.2 Ghz CPU and 1 GB of RAM. For the server, we used a quad-core desktop with a 3.1 GHz CPU and 8 GB of RAM running Ubuntu 11.10. To execute Android on a regular Intel x86 desktop, we built a target of Android for VirtualBox 4.1.8. We also used an off-board equipment [13] to profile energy consumption of the smartphone.

To evaluate the effectiveness our offloading model, we implemented three benchmark applications in different categories: CPU, IO, and user interactive. We also tested scientific and real applications to evaluate our partial stack transfer. In the following subsections, we describe our applications and its experimental results in detail.

2.4.1 CPU and IO Benchmark

As our CPU tasks, we chose a FIBONACCI sequence generator. The FIBONACCI sequence generator recursively calls its member method to generate a FIBONACCI sequence for a given size of a sequence. Because the performance of this application depends mainly on the CPU power of the device, we classified it as a CPU task. In Section 2.3.1, we also used three scientific kernels, Fast Fourier Transform (FFT), LU factorization (LU), and Successive over-relaxation (SOR) of SciMark 2.0 benchmark suite [14] to show the number of frames actually captured in their optimal offloading scenario[2].

As one of our IO tasks, we implemented the face detector, which recognizes all faces in a given image. After all faces are found, the detector draws green

[2]The experimental results for these kernels were already presented in Table 2.4.

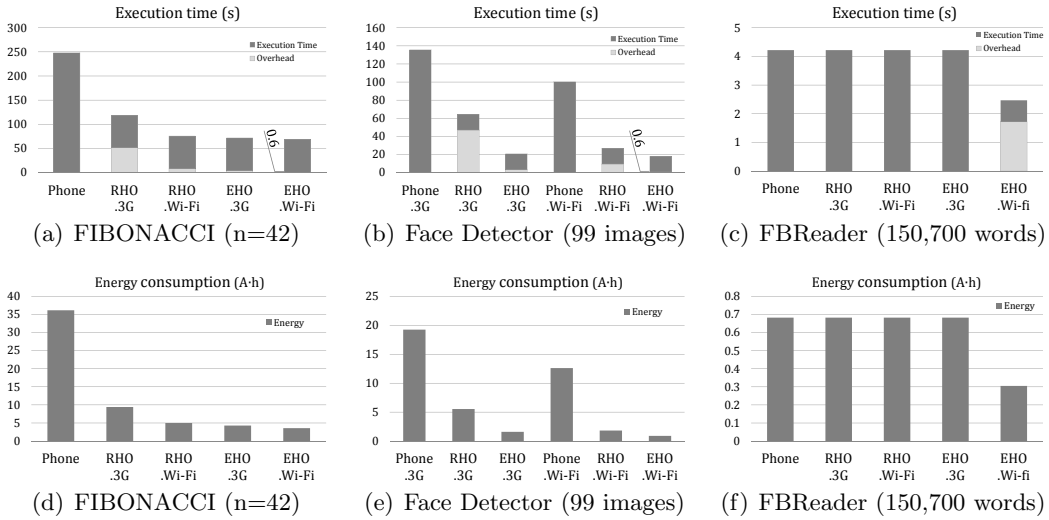
Table 2.5 Methods of the modified FBReader

Name	Description
selectBlocks	Selects blocks for a given book
searchWords	Searches 15 words for given blocks
searchBlockList	Searches one word for given blocks
searchBlock	Searches one word for a given block
find	Searches one word for a given paragraph

rectangles on the each face. We used OpenCV 2.3.1 library for Android to implement our detector. The detector downloads its input images from an external on-line server at run time. We chose it as a IO task, since the computation heavily relies on IOs involving network operations and file accesses.

Another IO task we chose is FBReader [15], which is a state of the art open source e-book reader application. To hasten its computation, FBReader loads an entire e-book into its memory space. This is the reason why we chose it as our IO task. Among the many functions of FBReader, we chose the word search to offload; for a given book and a word, it returns every position of the words in the book which has the same character pattern as the given word. We adjusted the code to automatically search 15 pre-given words in a part of the book at a time, instead of searching the entire book with a user-given word. The modified code searches the words through five steps which are shown in Table 2.5.

For the evaluation, we vary the size of the sequence between 25 and 42 for the generator, the number of images from 1 to 99 for the detector. We also vary the size of books from 6,420 words to 150,700 words for the FBReader. To profile our applications, we used a set of randomly generated inputs; for the FBReader, we randomly chose 15 words from a pre-built list [16]. By using the profiling result, we solved the partitioning problem in a similar way to CloneCloud [3], and built the partitioning scenario for each application. We



(a) FIBONACCI (n=42) (b) Face Detector (99 images) (c) FBReader (150,700 words)

(d) FIBONACCI (n=42) (e) Face Detector (99 images) (f) FBReader (150,700 words)

Figure 2.12 Average phone execution times and energy consumptions for FBReader, FIBONACCI sequence generator and face detector with the largest input size

also applied our static analysis based on the compiler technique to reduce the state transfer costs, which is mentioned in Section 2.2.

Figure 2.12 shows execution times and energy consumption of the smartphone for three applications on the largest input size. In Figure 2.12(b) and Figure 2.12(e), the measurement for phone-alone execution is divided to different bars ("Phone.3G" and "Phone.Wi-Fi") because our detector downloads its input images from an external on-line server at run time, therefore the network latency affects its execution.

For the largest input size of the face detector, we obtained a speedup of 6.6 on the smartphone over 3G, and 5.7 over Wi-Fi. Our approach achieved much higher improvement than the RHO approach, whose speedup is 2.1 over 3G and 3.7 over Wi-Fi. It is induced by reducing the state transfer time from 47 second to 3.7 second over 3G, and from 9.4 second to 0.6 second over Wi-Fi. We also improved the performance of the RHO approach by about 65% over 3G

Table 2.6 Evaluation result for another input size

Benchmark	RHO.3G(s)	RHO.Wi-Fi(s)	EHO.3G(s)	EHO.Wi-Fi(s)
FBReader (28,030 words)	N/A	N/A	N/A	1.564
FIBONACCI (n = 34)	N/A	N/A	3.215	2.023
Face Detector (30 images)	N/A	12.547	8.271	4.433

and 11% over Wi-Fi. Notice that our approach is more effective over 3G than Wi-Fi. We believe that such a result is caused by different network latency; due to the greater latency and lower bandwidth, the migration cost over 3G network is much higher than Wi-Fi. Similar to CloneCloud's result [3], energy consumption generally follows execution time. We also achieved similar results for the FIBONACCI sequence generator.

For the FBReader, our approach achieved a speedup of 1.7 over Wi-Fi, although the RHO approach failed to offload it. Similar results are shown in Table 2.6; the RHO approach failed to offload every application except the face detector over Wi-Fi. In contrast, our approach succeeded in offloading all except the FBReader over 3G. This result demonstrates that the reduction of the state transfer costs really has great impacts on the performance of execution offloading, as predicted earlier.

2.4.2 User Interactive Chess Game

Another benchmark that we tested is a *chess engine* which is a central part of a user interactive chess program. The engine accepts the user's move as the input, and returns a 'counter move' given the position of each chess piece on the board. To find the optimal counter move, it uses a simple minimax algorithm: it considers all possible next move and scores them by traversing a game tree, which is a directed graph whose nodes are positions of each chess piece and whose edges are moves. After that, it returns the highest scored move. If there are more than one moves whose scores are equal, the engine chooses one of them

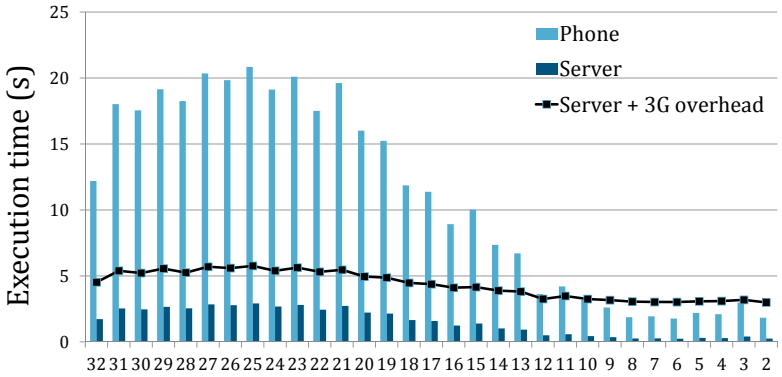


Figure 2.13 Average execution times of getNextMove for each distinct number of pieces left on the board.

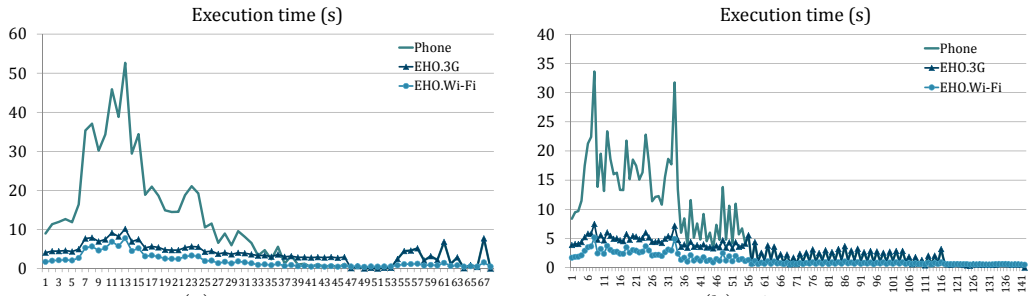
randomly.

To obtain the optimal counter move, our chess engine invokes a key REM, `getNextMove`. As stated in Section 2.1.1, the solver makes an offloading decision for the REM based on the profiling result which is, in the original implementation [3], the average of execution time and energy consumption of an application on a set of randomly generated inputs. Such a strategy might be acceptable for some applications whose execution times are relatively consistent regardless of the sequence of their input. According to our analysis, however, this simple strategy is not workable for others like our chess engine whose execution time drastically varies on its input values. To explain this, see from Figure 2.14 the execution times of our engine on the phone which are measured and plotted every time a user input is given. One noticeable thing here is the similarity between two curves of these time plots: the execution times rapidly hike as the matches start, but after reaching the top at the early stages, they both gradually drop as the matches come close to an end. Among various factors resulting in this execution pattern, a major one we found is the number of pieces left on the board. To show this, we display in Figure 2.13 the average run times of

`getNextMove` for each distinct number of live pieces. The figure evinces the similarity between the run time curve drawn in a decreasing order of the number of pieces and those in Figure 2.14.

This observation had led us to conclude that when using the profiled performance data to make offloading decisions for the REMs of our chess engine, the solver must consider the number of pieces currently alive for better performance. For instance, the execution times of `getNextMove` are on average 10.9 sec on the phone and 4.3 sec on the server (including the 3G communication overhead added for state transfer) respectively. Therefore in the original design, the solver may decide that offloading the REM over 3G is always profitable. This naive decision, however, will result in the performance loss for some cases like those with less than 10 remaining pieces where running the REM on the phone is clearly more profitable as shown in the Figure 2.13. Consequently in our new design, the profiler estimates the execution times of the chess engine REMs for each different number of pieces on the board, and the solver makes variable offloading decisions for the same REM depending on the number of pieces. In this experiment, we have 31 decision points for `getNextMove` over 3G and Wi-Fi, respectively. For each point, the migrator either offloads the REM or not at run time, following the decision produced by the solver.

In Figure 2.14, the performance results of our execution offloading strategy for the chess engine are presented. We played each game until 2 pieces were left. Each play consists of 68 and 142 moves, respectively. For the play with 68 moves, we obtained a speedup of 2.7 over 3G and 5.4 over Wi-Fi and for the play with 142 moves, 2.2 over 3G and 4.5 over Wi-Fi. Even if our variable decision strategy was sometimes incorrect, thereby causing occasional performance loss as in the cases of the 41st move in Figure 2.14(a) and the 45th move in Figure 2.14(b), it was proven to be correct for most cases, achieving overall sig-



(a) 68 moves (b) 142 moves

Figure 2.14 Average phone execution times of the chess engine on two plays. Time unit is second.

Table 2.7 Energy consumption of the chess engine on two plays.

Play	Phone(A*h)	EHO.3G(A*h)	EHO.Wi-Fi(A*h)
68 moves	122.39	107.98	28.44
142 moves	165.05	169.29	71.72

nificant performance gains from execution offloading. The energy consumption of the chess engine on two plays is also shown in Table 2.7, demonstrating the effectiveness of our strategy in terms of energy saving[3].

2.4.3 Impact of Partial Stack Transfer

Including the optimal scenario used in Figure 2.12, we tested four offloading scenarios for FBReader to evaluate the impact of the partial stack transfer. We fixed the migration point of these scenarios to the entry of `find` and varied the return point of each scenario as shown in Table 2.8. We also built and evaluated two scenarios in a similar way for each benchmarks, fibonacci, chess engine and face detector; while the first scenario includes every MBM[4], the second one

[3]Note that the energy consumption of offloading is slightly higher than the local execution for the play with 142 moves over 3G; It is due to the network latency occasionally fluctuating at run time, which led to the communication module of the phone spending more energy to connect itself to the network.

[4]So, it is exactly same with full stack transfer.

Table 2.8 Offloading scenarios for FBReader

Name	Migration point	Return point
selectBlocks	entry of `find`	exit of `selectBlocks`
searchWords	entry of `find`	exit of `searchWords`
searchBlockList	entry of `find`	exit of `searchBlockList`
searchBlock	entry of `find`	exit of `searchBlock`

finished only a part of them in remote execution.

Figure 2.15 shows the impact of partial stack transfer on the size of the transferred state for each scenario. For `searchWords`, the optimal scenario of FBReader in Figure 2.15(a), the partial stack transfer drastically reduced the size of the state; 52% on the largest sized book and 39% of on the middle sized book. For the other scenario `searchBlock`, the size of the state is reduced even more; 94% on the largest sized book and 70% on the middle sized book[5]. We believe that such a result is caused by the nature of FBReader; because each method in Table 2.5 accesses a different range of its input book in memory space, the range of memory space accessed by each scenario is also different in respect to the methods making up the scenario. For example, the method `searchWords` accepts "blocks", which is a part of a book, while the method `selectBlocks` accepts the entire book as its input. Therefore, instead the entire book, the scenario `searchWords` needs only some blocks in remote execution. This is why the partial stack transfer dramatically reduced the size of the state for `selectWords` and `searchBlock`. We also reduced 24.3% to 27% of the state for the chess engine; similar to FBReader, stack frames excluded by `scenario1` in Figure 2.15(b) possessed lots of heap objects, which were not necessary in remote execution any more.

[5]Nevertheless, `searchWords` is still the optimal scenario for FBReader. Because the execution time of `searchBlock` in remote execution is too short, so choosing `searchBlock` is not profitable even though the partial stack transfer dramatically reduced the state of it.

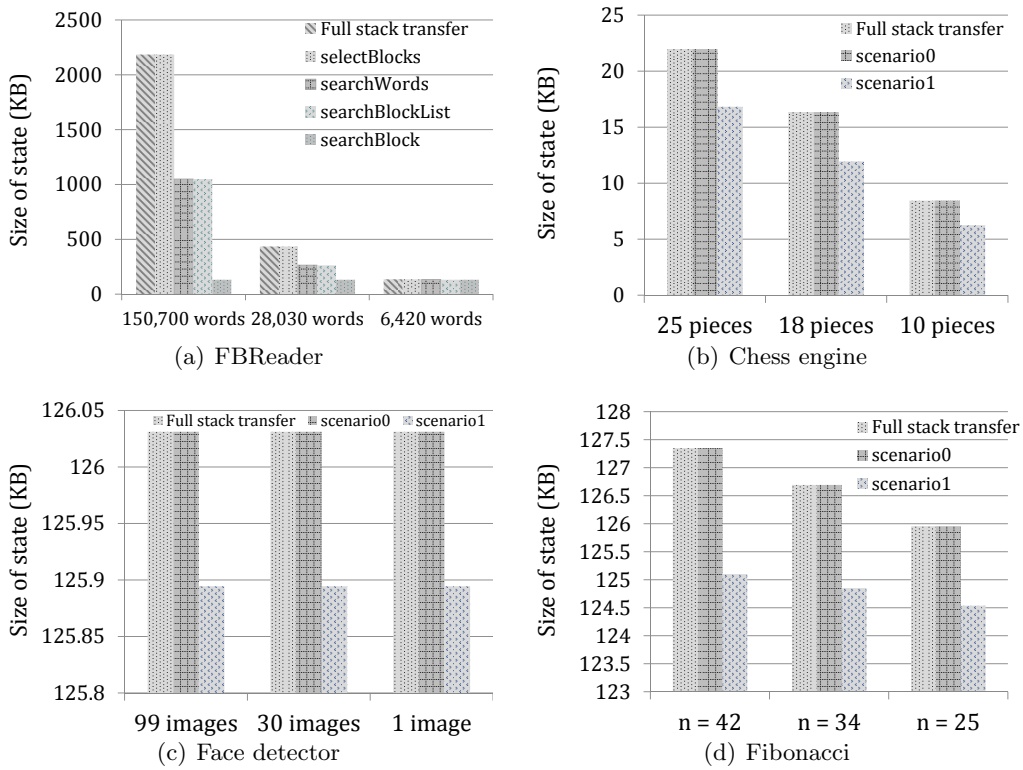


Figure 2.15 Impact of partial stack transfer on the size of the transferred state.

Contrary to the impressive results for FBReader and chess engine, the effectiveness of the partial stack transfer is ignorable for the face detector in Figure 2.15(c). This is because most heap objects are used by its core REM, which downloads an image and searches faces, and the caller method of the REM only used a few bytes of local variables. Similarly, only 1.2% to 1.8% of the state are reduced for the fibonacci generator. For `scenario1` in Figure 2.15(d), the partial stack transfer excluded half of the stack frames which are recursively pushed into the call stack; because each method equally used small amounts of local variables, however, such a optimization was not quite effective.

2.5 Discussion

Our proposed techniques in this paper are tailored to CloneCloud, which captures the existing call stack with heap objects. Indeed, liveness analysis and partial stack transfer are closely related to stack transfer; these techniques are not necessary where the stack is not transferred. However, we believe that the key concept of EHO can be widely adopted by other offloading approaches, where the state of an application is captured and transferred. To serialize only the essential state for its "remoteable" method, for example, MAUI [2] can adopt the concept of EHO even though it does not transfer any stack frames. In such cases, the information generated by our analysis and the mechanism which uses the information should be adjusted appropriately to the approach.

We also designed our techniques to work on Dalvik VM, which has its own memory layer. It enables us to interpret and settle the transferred state on remote sites; for this reason, we believe that our techniques can be easily applied to .NET runtime, which has its own memory management layer, or any other framework whose nature is similar to .NET or Dalvik VM.

As we mentioned in earlier sections, we used conservative approaches for our techniques. To cover every possible flow in remote execution, for example, we used static call graphs instead of dynamic call graphs. We believe that such conservative approaches help us avoid misprediction entirely. However, if the execution flow or memory usages in remote execution are predicted[6] for more aggressive optimizations, other techniques should be required to prevent or correct misprediction.

[6]For example, dynamic call graphs can be used instead of static call graphs.

2.6 Related Work

One of the earliest studies that aim to empower portable devices with surrounding servers was done by Satyanarayanan et al. [7], who have developed versions of *ISR* systems for the past decade. To offload a process running on the device, they migrated the full VM or OS image along with the process. Not surprisingly, the amount of transferred data for *VM migration* tends to be huge (around in the order of gigabytes). To lighten the load, they proposed the *dynamic VM synthesis* approach [5] where a small VM overlay is sent by a mobile device to the *cloudlet* (nearby small cloud) that is already installed with the base VM which the overlay was derived from. The overlay size was reported about one order of magnitude smaller than the full VM size, so they claimed that the approach might be feasible for mobile computing using fast wireless LANs like Wi-Fi. However, even that figure would be still too high for lower bandwidth WAN interfaces like 3G.

In order to make mobile cloud computing more viable over the wireless WANs, many recent studies listed below have proposed *process-level migration* approaches that normally require only a few megabytes [2] for each state migration. These approaches can be divided largely into two groups: those using static partitioning schemes and those using dynamic ones. A noticeable work in the first group might be Wishbone [8], which gives a solution for optimal partitioning of sensor network application code across sensors and servers. It statically partitions the application code based on profile data that include the computational and network load by using an integer linear program to find the minimum use of CPU and network bandwidth. Wishbone guarantees that the optimal partitioning can be predetermined regardless of the target hardware platform because it targets a confined area of applications where a division of

subtasks is fairly clear.

In static partitioning schemes, using the programming models provided by the middleware or APIs, the users must manually specify at compile time how to partition their code, what state to migrate, and how to adjust the offloading strategy to the varying network status. For example, Cuckoo [17] offers their programming model to help users make their applications offloadable. To relive users from such burdens, a majority of studies have been interested in dynamic or semi-dynamic partitioning schemes. One of the first ones is OLIE [18], which collects the current status of the memory utilization and available network bandwidth to decide whether offloading should be triggered at run time. But the main goal of OLIE is to overcome only the memory resource constraints of mobile devices. This is deemed relatively simple as compared to optimizing energy consumption and execution times, which is our goal like others [2, 3]. As another example, Odessa [6] dynamically partitions applications using a greedy algorithm, and adaptively makes offloading decisions. However, the developer who tries to apply their approach must use the specific development framework. Giurgiu et al. [9] propose an elaborate system that dynamically distributes several components of an application between a server and a smartphone. The system is realized on top of their middleware that can support the actual distributed deployment of an application between machines. However, the application must be coded in a special language in order to be worked with this approach, while we support ordinary Java.

CloneCloud [3] suggests dynamic execution offloading approach by modifying the mobile execution environment, Dalvik VM, to capture the current execution state. CloneCloud can reduce the run time overhead, because they do not need to modify the application code while some approaches have to do. Some approach appends new statements to the application's code to do

that, because they do not want to modify the execution environment to keep the flexibility. Due to the new appended code, a significant overhead may be incurred on applications performance [4]. There are some approaches such as MAUI [2] that labor to reduce the runtime overhead. MAUI is a RPC based offloading architecture which decides at run-time which methods should be remotely executed based on the best energy savings possible under the mobile device's current connectivity constrains. MAUI requests user annotation on the application code to mark migratable methods. Ma et al. [4] suggest a Java bytecode transformation technique to migrate computation from a mobile device to a server based on Java exception handling mechanism without imposing significant overhead on normal execution. But it still has much overhead when migration is taken place.

More recently, several approaches have been proposed to improve the performance of execution offloading. ThinkAir [19] suggests a dynamic resource allocation scheme, which allocates more than one clone VM for the offloaded application to exploit parallelism and relieve the lack of memory space. By adopting distributed shared memory into its offloading framework, COMET [20] expands the range of offloadable code and consequently, allows multiple threads to be offloaded simultaneously. Inspired by MAUI, Kovachev et al. [21] present their middleware which serves more sophisticated profiling, monitoring and partitioning decision. Although these work have their own contributions, none of them explicity discusses how to use compiler static analysis to reduce the amount of migrated state.

Chapter 3

Cloud Platform for Cost-Effective Execution Offloading

In this chapter, We introduce *CMcloud*, a novel cost-effective mobile cloud platform, which works nicely under the real-world cloud environments. The key idea of CMcloud is to exploit a novel performance modeling methodology for estimating the target application's post-offload performance accurately on any target server, regardless of its current utilization. At the same time, CMcloud allows to offload as many applications to each server as possible without violating the applications' user-expected performance. If the target performance cannot be achieved using the currently allocated server due to inaccurate performance estimations, CMcloud performs fast inter-server live migrations to achieve the target performance. In this way, CMcloud can offer to users its QoS-guaranteed offload service at a very low price, while minimizing the cloud operation costs.

CMcloud operation assumes the following working environments. First, CMcloud is given the target application's performance profiled on both the user device and a reference-model cloud server. Such static profiling assumption of

CMcloud is similar to that of existing offload schemes [2, 3], and thus it does not incur any extra profiling overheads compared to the existing schemes. Second, CMcloud allows to run as many applications on each server as possible to minimize the cloud operation costs.

Based on the environments, CMcloud works as follows. First, on receiving an offload request, CMcloud applies a sophisticated architecture performance modeling to find the most cost-effective target server whose remaining resources are just large enough to achieve the target performance. CMcloud finds the most cost-effective target server by accurately predicting the application's performance by estimating how the performance profiled on the reference server would change on the target server, regardless of its current utilization. Next, CMcloud performs offloading and starts to monitor the application's progress. If CMcloud detects any failure in achieving the target performance due to either inaccurate estimations or unexpected performance contentions, it performs inter-server live migrations to achieve the target offload performance. In this way, CMcloud provides the most cost-effective offloading service to users without violating the QoS of the offloaded applications.

To the best of our knowledge, CMcloud is the first mobile cloud platform to provide the cost-effective offloading service by taking into account the costs of cloud operation and the quality of offload services. Our example implementation on top of a 8-node (16 sockets) Android-x86 / KVM [22] with QEMU 1.4.0 / Ubuntu 12.04 64bit platform shows that CMcloud can improve the server throughput by 84% over a conventional static light-load scheme (or a 2.7x per-socket throughput.) Alternatively, CMcloud reduces the number of service failures by 83% over a static high-load scheme, while even improving the throughput by 31%.

Our work makes the following contributions:

- **Novel design.** We propose CMcloud, a novel cost-effective mobile cloud which exploits a performance modeling theory and inter-server migration capability.
- **High performance.** CMcloud significantly improves the server throughput over the conventional static load schemes (e.g., 2.7x per-socket throughput.)
- **Low costs.** CMcloud maximizes the server throughput or minimizes the server costs, while guaranteeing the user-expected offload performance.
- **Easy applicability.** CMcloud requires only a single reference-machine profiling to find the most cost-effective server, regardless of its current utilization.
- **Strong results.** Our results show that CMcloud can achieve 31% higher throughput over a heavy-load scheme, while reducing 83% of service failures.

3.1 Backgrounds and Limitations

To motivate our CMcloud platform, this section introduces conventional offload schemes and their key limitations.

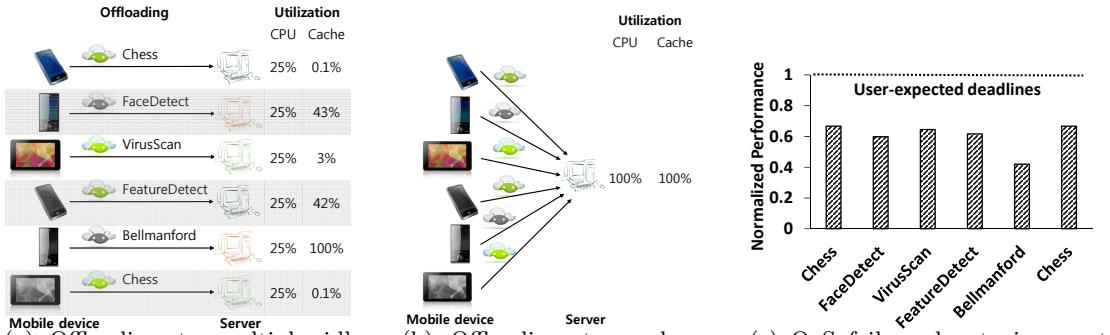
3.1.1 Basic Offload Mechanisms

The recent seminal works on the mobile-to-cloud offloading [2, 3] propose to run mobile applications on high-performance servers. Even though their detailed implementations can differ based on the code modification scope (e.g., user application, kernel,) and the offload granularity (e.g., functions, threads,) they are generally implemented as follows. First, the cloud provider must have profiled the target application's performance and power consumption on both the mobile device and the target server. Next, on receiving an offload request, the cloud provider compares the application's profiled performance on the mobile device and the target server. If any performance improvement is expected, which is likely to be the case unless the communication latency becomes an obvious bottleneck, the cloud provider offloads the application to the target server, and moves it back to the mobile device after the user-specified execution region is completed.

3.1.2 Limitations of Existing Schemes

However, as the existing schemes do not consider the user's service purchasing costs nor the cloud provider's server operation costs, they cannot be applied to the real-world cloud environments, where the cloud provider aims to maximize the server throughput or to minimize the server costs and charges the users based on their cloud resource usage.

Costs of offload services. The existing schemes completely ignore the costs of offload service by assuming that servers are provided for free and they



(a) Offloading to multiple idle servers

(b) Offloading to a busy server

(c) QoS failure due to incorrect profiling

Figure 3.1 Limitation of the existing mobile-cloud offloading schemes

run only one mobile application or maintain a same static load per server. Therefore, they always perform offloading as long as any amount of performance improvement is expected, which is likely to be the case because a lightly loaded server is available and runs faster than a mobile device. Figure 3.1(a) shows a typical scenario in which each four-core server accepts only a single offload request to achieve the highest performance and guarantee the user-expected performance.

However, the real-world clouds are designed to run as many applications as possible on each server to maximize the server's throughput or to minimize the number of active servers [23]. Therefore, if the existing schemes run only a small static load per server, the costs of operating the server and thus the user service fee will be significantly increased, which makes the mobile cloud computing business infeasible. Figure 3.1(b) shows a scenario in which multiple offload requests are serviced on a single server with a tradeoff between the server utilization and the offload performance.

Costs of service failures. To reduce the costs of operating the cloud and the user service fee, the cloud provider must allow to offload as many

mobile applications to each server as possible. However, offloading too many applications to each server incurs a new challenge in guaranteeing the user-expected offload performance because multiple applications come to contend for the sharing server resources such as cores and caches.

We define the number of offloaded applications completing within the user-expected deadline over the number of all offloaded applications as the offload service's quality of service (QoS). It should be noted that even a small QoS violation is unacceptable in the cloud business, as the users only pay the fee as long as the expected performance is achieved. Figure 3.1(c) shows a scenario in which five applications in Figure 3.1(a) are now offloaded to a single four-core server and all applications fail to complete within the user-expected deadlines. In this case, five applications contend for four cores and the last-level cache (LLC) available on the server.

Costs of profiling. The existing schemes assume that performance has been previously profiled for the target server and the offloading always achieves the profiled performance. However, this assumption is broken when an application is now offloaded to a target server which is running other applications to reduce the server costs. To enable an accurate performance estimation, the existing schemes must have profiled for all possible load states of each server. However, it is unrealistic for the cloud provider to statically profile every application for all possible server load states.

3.2 CMcloud Offloading

In this section, we first describe CMcloud's key design goals. Next, we present its basic operation model and architecture model consisting of three key components.

3.2.1 Design Goals

CMcloud must satisfy the following design goals to enable a cost-effective offload service. First, CMcloud must target a real-world commercial cloud environment, where servers are highly utilized by running multiple applications per server, Second, the cloud provider must be able to find the most cost-effective target server whose remaining resource is just large enough to achieve the target performance, regardless of its utilization. Finally, once an application is offloaded to the cloud, CMcloud must deliver the user-expected performance by considering the QoS success as a primary requirement.

3.2.2 Operation Model

Figure 3.2 illustrates how CMcloud performs an offloading once a user agrees to purchase the offload service. Therefore, the cloud provider now has a target deadline for each application to be completed by also considering a variation in the mobile-to-cloud transfer latency. The cloud provider must satisfy the deadline using the minimum server resources.

(1) Profiling on a reference server. CMcloud chooses a reference-model server in the cloud which is used to profile all offload-enabled mobile applications. Any server can be chosen as a reference-model server as long as it is equipped with a basic set of performance counters. CMcloud profiles the application's execution when the reference-model server is idle, and stores the information in the profiling DB. It should be noted that the same applications

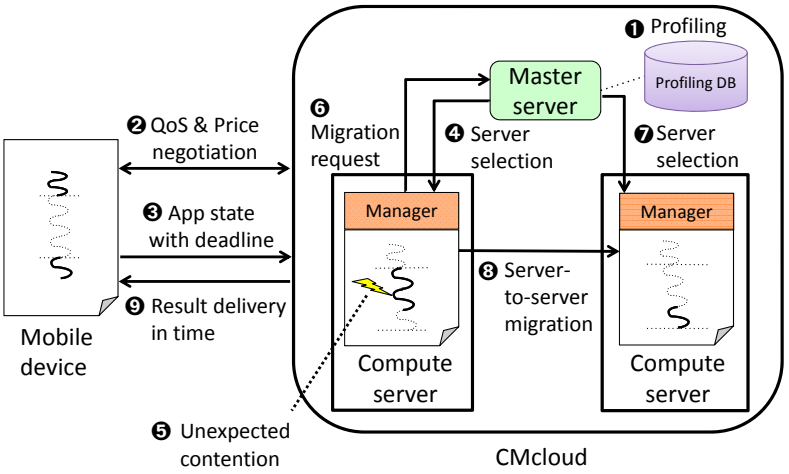


Figure 3.2 CMcloud's example operation model

with different inputs are considered as independent applications, as also proposed by [2, 3]. Section 3.3.1 describes the profiling mechanism in more detail.

(2)–(3) Offloading the application. The user requests an offload service, agrees on the service fee, and transfers the application with the termination point and the target deadline. Section 3.2.3 describes CMcloud's mobile-to-cloud offload mechanism in more detail.

(4) Selecting a target server. The cloud provider finds the most cost-effective server to complete the application within the target deadline using the minimum amount of resources. At this step, CMcloud applies a performance modeling methodology to estimate the application's performance on the target server by differences in server specifications (e.g., clock frequency, cache size) and load states between the reference-model server and the target sever. Section 3.3.2 describes the modeling mechanism in detail.

(5) Detecting a QoS failure. While running the application, the target server monitors the application's progress to detect a potential failure of completing the application within the target deadline, due to either an unexpected

performance contention or inaccurate performance estimation. Section 3.3.3 describes the monitoring mechanism in detail.

(6) Migrating to another server. On detecting a potential QoS failure, CMcloud accelerates the application by migrating it to a faster server. Section 3.3.3 describes the performance monitoring mechanism in detail. Section 3.3.4 describes the server-to-server migration mechanism in detail.

(7)–(8) Migration server selection. Similar to the step (2)–(4), the cloud provider selects the best target server based on the cost effectiveness and migrates the application to a new server. The cloud provider can repeat the steps from (5) to (8) to maximize the server throughput, while satisfying the QoS requirement.

(9) Completion. On reaching the offload termination point, the application is migrated back to the mobile device.

As a result, the user always achieves the expected performance for the paid service fee, while preserving the mobile device's battery. At the same time, the cloud provider can increase the server utilization to reduce both the datacenter operation costs and the offload service fee.

3.2.3 Architecture Model

In this section, we describe our CMcloud architecture, which consists of a single *master server* and the rest of servers as *compute servers*, as shown in Figure 3.3.

Master server. The master server consists of three components: *profiling DB*, *performance estimator*, and *target selector*. First, the profiling DB contains the profiled execution information on all offload-enable mobile applications on the reference-model server. Next, the performance estimator predicts the application's performance on a current candidate target server analyzing the profiled information on the reference-model server, and differences in server specifications and utilizations between the reference-model server and the candidate target server. Finally, the target selector finds the most cost-effective target server which will deliver the user-expected performance at the minimum costs.

Compute server. The compute server consists of three components: *manager*, *performance monitor*, and *migrator*. First, the manager communicates with other components and servers by handling requests and replies. Next, the performance monitor measures the application's on-going performance to detect a potential QoS failure (i.e., failing to meet the user-requested deadline) by exploiting the current server's performance counters and the execution profile stored in the profiling DB. Finally, on detecting a potential QoS failure, the migrator embedded in the application virtual machine (VM) suspends the application's execution, migrates its execution state, and continues to execute on a new target server.

Offload-ready mobile device The user's mobile device and operating system must be able to offload a mobile application to the cloud. In this work, we implemented a MAUI-like model as proposed in [2]. For example, the offload handler predetermines offload-enabled regions as remote-executable methods

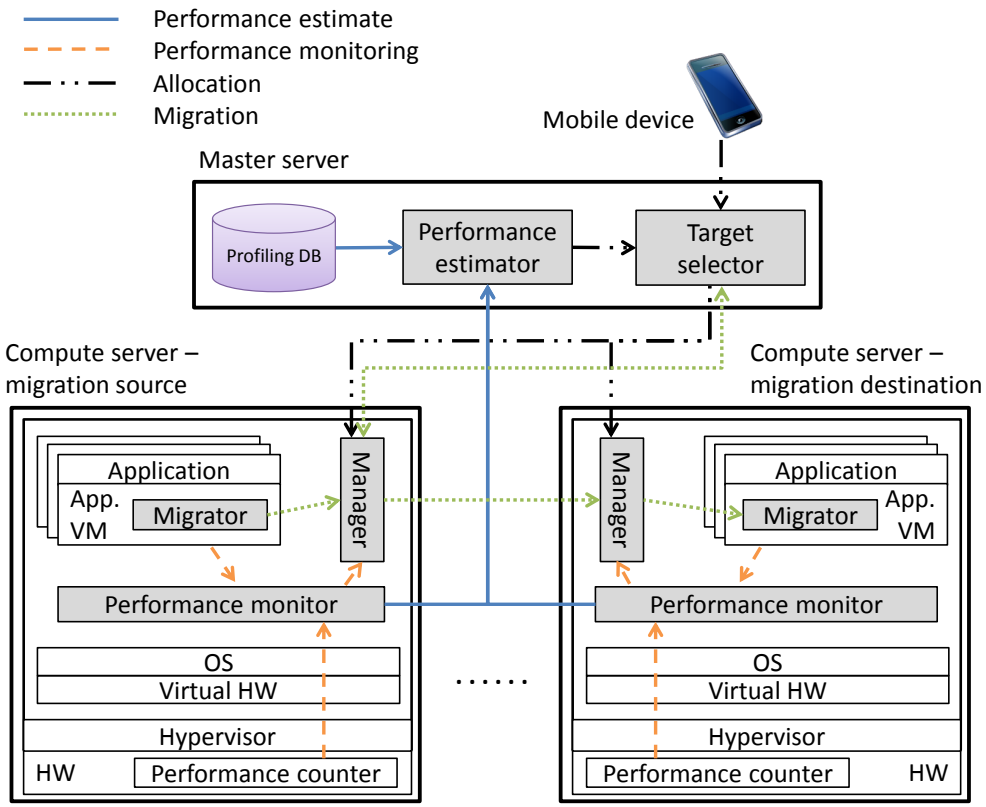


Figure 3.3 CMcloud's basic architecture model.

(RM). Therefore, the master server must profile the RM methods and store the profiled information in the profiling DB. Even though we used a MAUI-like model for this work as it does not require to modify the operating system, CMcloud implementation is orthogonal to the mobile-to-cloud offload implementation. CMcloud focuses on providing the cost-effective cloud platform. Therefore, CMcloud can be implemented with other mobile-to-cloud offload models.

Network modeling We modeled 3G and Wi-Fi networks between mobile devices and the cloud using normal distributions of the bandwidth with empirically observed average and deviations. The detailed information is described in Section 3.4.

3.3 CMcloud Mechanism

In this section, we describe CMcloud operation mechanisms in detail: reference-server profiling, performance estimation and monitoring, and migration techniques.

3.3.1 Reference-model Server Profiling

The existing offload schemes assume that the offloaded application's performance has been previously profiled for the target server so that they can estimate the application's post-offload performance before making an offload decision. However, if the target server runs different sets of applications from the profiling time, which is the basic operation model of CMcloud, the existing schemes must perform an unbounded number of profiling processes for all kinds of different utilization status even for a single server.

On the other hand, CMcloud still performs a static profiling on a single reference-model server, which can be later translated to the performance for a different target server running any combination of applications. To enable

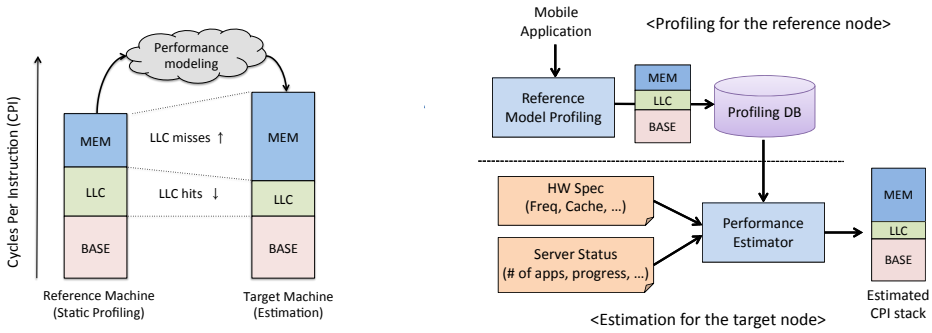
such performance estimation, CMcloud collects the following statistics on the reference-model server using HW performance counters and a memory access tracer.

- **CPI stack.** Execution time breakdown to each performance bottleneck component (Section 3.3.2.)
- **Temporal locality information.** Memory access patterns affecting cache hit and miss rates (Section 3.3.2.)
- **Runtime progress.** Performance progress information collected per second (Section 3.3.3.)

CMcloud can choose any machine equipped with basic performance counters as a reference-model server. However, as our performance modeling assumes that the server's pipeline microarchitecture (e.g., branch predictor, issue order) is maintained, CMcloud must profile an application on all reference-model servers representing unique pipeline microarchitecture families (e.g, one reference machine for all Sandy Bridge family processors.) Other than the pipeline structure, CMcloud does not require extra profiling due to different clock speeds or different sizes of last-level caches (LLC). More importantly, CMcloud does not require extra profiling due to different server utilization status. Therefore, CMcloud's static profiling overhead is much smaller than that of existing offload schemes [2, 3] required to estimating the post-offload performance when severs are highly utilized.

3.3.2 Performance Estimation

In this section, we explain how CMcloud estimates the performance for a different server with different utilization, based on the profiled information using the reference-model server.



(a) Performance estimation using CPI stack

(b) Performance estimation process

Figure 3.4 CMcloud's performance estimation process using architecture performance modeling

Performance Analysis using CPI Stack

CPI stack [24, 25] is a performance analysis tool widely used to understand how much each performance losing events (e.g., cache miss, branch misprediction) contributes to the overall performance. As cycle-per-instruction (CPI) explains how many cycles are spent to execute a single instruction on average, it is possible to separate the different impacts from different bottlenecks. If the CPU experiences performance losing events such as a cache miss, the final CPI can be obtained by adding the ideal CPI and the extra CPI caused by the cache miss. Therefore, if we are aware of how each event's CPI impact would change on a target architecture, it is possible to construct a target CPI, as shown in Figure 3.4(a).

Estimation for different idle servers

CMcloud applies the CPI stack method to predict the target application's post-offload performance on a target server, using the profiled performance on the reference-model server. CMcloud first takes the CPI stack collected on the reference-model server, analyzes how key performance losing events will change

on a target server, and constructs a new CPI stack to measure the post-offload performance, as shown in Figure 3.4(b).

In this work, CMcloud focuses mainly on four performance impact factors, CPU frequency, LLC hit, LLC miss, and store buffer full, because the number of memory instructions and LLC miss rates affect the overall performance most significantly. Even though we consider only four major performance factors in this work, CMcloud can apply more fine-grain bottleneck components as proposed in [25, 26, 27, 28].

Once such CPI stack becomes available, CMcloud can estimate the performance on a target server by adjusting the impact of each CPI stall event as follows. First, CMcloud breaks the overall CPI down to a combination of four sub-CPI events (i.e., ideal latency ($base$), last-level cache hit (llc), memory access (mem), store buffer full ($sfull$)) as follows.

$$CPI = CPI_{base} + CPI_{llc} + CPI_{mem} + CPI_{sfull} \tag{3.1}$$

Next, CMcloud measures CPI adjusting factors, $CPI_{ratio,mem}$, $CPI_{ratio,llc}$, and $CPI_{ratio,sfull}$. The factors are used for adjusting the corresponding CPI event for the target server.

If the CPU clock frequency of the target machine is different from that of the reference server, both CPI_{llc} and CPI_{mem} are scaled for the target CPU. If the target server's memory access latency is different from that of the reference server, the ratio is applied to CPI_{mem} as well.

$$\begin{aligned} Freq_{ratio} &= Freq_{target}/Freq_{ref} \\ CPI_{ratio,llc} &= LLC\_Hit_{ratio} \times Freq_{ratio} \\ CPI_{ratio,mem} &= LLC\_Miss_{ratio} \times Freq_{ratio} \times Lat_{ratio} \end{aligned} \tag{3.2}$$

where $Freq$ is a CPU clock frequency, Lat is a memory access latency, and LLC_Hit and LLC_Miss are the number of LLC hits and misses, respectively.

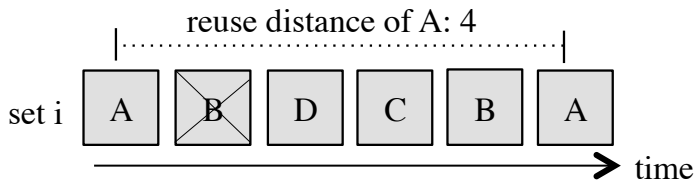


Figure 3.5 An example reuse distance of four for A.

However, it is difficult to estimate the number of LLC hits and misses, when cache architectures differ between the reference server and the target server. To address the issue, we assume that the size of an LLC differs by the degree of associativity and its cache block replacement policy is based on a LRU policy. In fact, as modern LLCs exploit variations of index hashing mechanisms to effectively increase the degree of associativity, even caches scaled by the number of sets show similar hit and miss patterns as the caches scaled by the associativity.

To discover the application's temporal locality, we leverage the reuse distance (RD) analysis[29], in which RD is the number of distinct and different memory accesses between two consecutive references. Figure 3.5 shows an example reuse distance of four between two consecutive memory accesses to A. To collect the reuse distances, we use our memory tracing scheme implemented in the QEMU emulator.

With the LLC scaling and the reuse distances available, LLC hit and miss rates can be estimated for differently sized LLCs. For example, when the LLC's associativity increases from x to y, the number of LLC misses decreases by $\sum_{n=x+1}^{y} C_{RD=n}$ where $C_{RD=n}$ is the number of accesses with the reuse distance

of n. Therefore, LLC_Miss_{ratio} can be calculated as follows:

$$LLC\_Miss_{ratio} = 1 \pm \sum_{n=x+1}^{y} C_{RD=n}/LLC\_Miss_{ref} \tag{3.3}$$

Finally, the penalty caused by the store buffer full depends on some factors including the issue width (W), in-flight store instructions, memory latency and clock frequency. Frequent LLC misses of store instructions can incur a high penalty by filling up the store buffer, which stalls the entire piepline. We estimate such store buffer full cycles using the measured CPI_{sfull} and average store instructions per cycle. We approximately calculate the changed penalty of store buffer full event as follows.

$$CPI_{ratio,sfull} = 1 + \frac{Freq_{ratio} \times Lat_{ratio} - 1}{W \times \%stores} \tag{3.4}$$

By combining equations for each CPI event, we obtain the final target CPI estimation model for different, but idle target servers:

$$CPI_{target} = CPI_{base} + \sum (CPI_{ratio,event} \times CPI_{event}) \tag{3.5}$$

Estimation for Different Utilization

In highly utilized cloud environments, each server is highly utilized to achieve the maximum throughput, and thus it will be difficult to find an idle target server for offloading. If an application is offloaded to a target server currently running other applications, the available CPU clock cycles and LLC capacity will be smaller due to the resource sharing among applications. To calculate the available clock cycles with a core contention, we simply scale the baseline frequency down by the number of applications. We assume that all applications are evenly scheduled with same priorities. If the operating system applies different priorities, this method can be easily adjusted to consider the relative weights as cycles available.

In addition, to estimate the miss rates of the LLCs experiencing a contention, we exploit the miss rate estimation model as proposed in [30]:

$$LLC\_Misses = C_{RD>A} + \sum_{x=1}^{A} P_{miss}(x) \times C_{RD=x} \tag{3.6}$$

where A is the LLC's associativity, $C_{RD=x}$ is the number of accesses with the reuse distance of x, and $P_{miss}(x)$ is the possibility of miss for the access with the reuse distance of x.

P_{miss} depends on which applications are co-located in the same server. This estimation requires the histogram information such as per-application reuse distances. In our work, as the phase of each application varies over time, we collect the information periodically (e.g, one billion instructions.) Then, we adjust the LLC miss estimation model by considering progresses of background applications in the server where a new application is offloaded.

Once such information becomes available, we apply the modified frequency and miss information to the formulas developed in the Section 3.3.2.

3.3.3 Performance Monitoring

In this section, we describe CMcloud's performance monitoring mechanism. The monitoring mechanism detects the applications' potential QoS failure caused by either an incorrectly estimated post-offload performance or a resource contention in the servers.

Performance Evaluation

The *performance monitor* shown in Figure 3.3 exploits hardware performance counters to check the progress of the target application. Our implementation collects the million instructions per second (MIPS) of each application using a modified version of `perf` [31]. Based on the performance estimation model described in Section 3.3.2, the performance is periodically measured and compared as the number of retired instructions for the given period (e.g., one second.)

QoS Violation Detection

The performance monitor detects a QoS violation as follows. First, as CMcloud profiles applications only on a single idle reference-model server, the performance monitor estimates the expected performance on the current target server using the model described in Section 3.3.2. Next, the performance monitor periodically compares the application's target MIPS and the profiled MIPS. Figure 3.6 describes the QoS violation detection method as follows:

(1) Determine a comparison period. The performance monitor determines a small period of region (e.g., three past seconds) to compare the MIPS. We use few-second comparison periods to tolerate sudden fine-grain performance variations.

(2) Find the same period for the expected progress. The performance monitor finds the corresponding period from the expected progress, and then

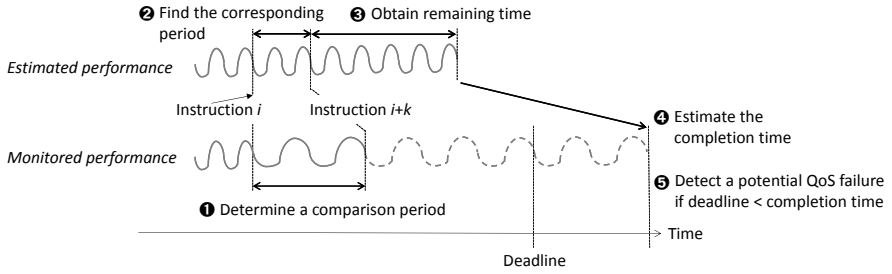


Figure 3.6 Performance monitoring.

measures the relative performance difference during the period.

(3) Obtain the original completion time. The performance monitor computes the time spent to complete the application based on the originally estimated post-offload performance.

(4) Compute the newly expected completion time. By applying the relative performance difference between the expected post-offload performance and the currently monitored performance, the performance monitor can estimates the application's expected completion time.

(5) Detect a QoS failure. The performance monitor can now detect a potential QoS failure by comparing the newly expected completion time against the target deadline agreed between the user and the cloud provider.

3.3.4 Migration

On detecting a potential QoS failure, CMcloud guarantees the application's QoS requirements by migrating the corresponding applications to a faster server. The *migrator* shown in Figure 3.3 performs a low-cost live VM migration.

Destination selection. On detecting a QoS failure, the migrator must find a right destination server. When a migration request is forwarded to the target selector, the target selector finds a right destination node using the performance

estimator, as described in Section 3.3.2. The performance estimator exploits not only the status of each server (e.g., the number of active cores, current server utilizations,) but also the application-specific information (e.g., the number of retired instructions, the elapsed run time.)

Performance overhead. Migration can incur non-trivial performance overhead when the large amount of data is transferred over the network. Therefore, CMcloud performs fast inter-server live migrations to minimize a downtime. We assume that servers already contain key application binaries to avoid migrating binaries.

3.3.5 Cost-aware Application Scheduling in Cloud

To minimize the datacenter operation costs, CMcloud targets to improve server utilizations, while maintaining only a smallest number of active servers in the cloud. To achieve the goal, CMcloud first starts with a small number of nodes and populates the small pool with offloaded applications. Next, on receiving a mobile-to-cloud offload request, the performance estimator collects the estimated performance from the servers. Using this information, the target selector finds the most cost-effective server whose remaining resources are just enough to satisfy the agreed post-offload performance. If the target selector cannot find such server, a new server is activated and added to the current pool of active servers.

Table 3.1 CPUs used for tests.

	Processor	Frequency	Cache Size
Reference	Intel Core i7-930¶	2.80 GHz	8 MB
	Intel Core i7-2600‡	3.40 GHz	8 MB
Target	Intel Xeon X5650¶	2.66 GHz	12 MB
	Intel Xeon E5-2630‡	2.30 GHz	15 MB
	Intel Xeon E5-2670‡	2.60 GHz	20 MB

¶Nehalem (Westmere) processor
‡Sandy Bridge processor

3.4 Evaluation

In this section, we first explain our evaluation platform and workloads, and next evaluate CMcloud's accurate performance modeling and its overall cost effectiveness.

Server platform. Our datacenter consists of eight server nodes connected with 10Gbps network, where each server has two CPU sockets. All servers run Ubuntu 12.04 64-bit with Linux Kernel 3.5.0 and KVM [22] with qemu 1.4.0. The KVM release supports both hypervisor and users to access low-level performance counters. To support offloading between mobile phones and x86 severs, we use Android-x86 [32] VMs to run an Android application on a server.

Table 3.1 lists CPU architectures used as reference-model and target servers. We use reference models for different pipeline micro-architecture CPU families (e.g., Nehalem, Sandy Bridge) to avoid inaccurate performance estimation across different micro-architectures. As a result, we use two unique reference CPU models in this work because the target servers use one of the pipeline architectures, but differ in the clock frequency and the cache size.

Network. We modeled a Wi-Fi network using a normal distribution of the bandwidth with empirically obtained 18.5Mbps average and a 3.5 standard deviation. Each offload request obtains a unique bandwidth following the distri-

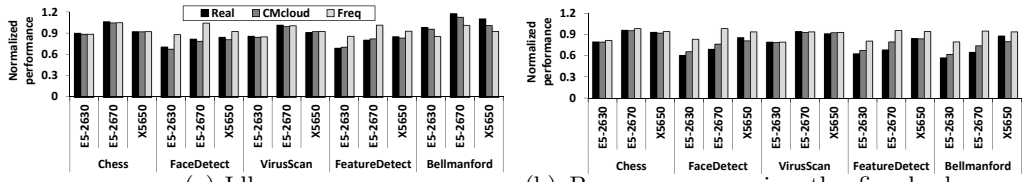
Table 3.2 Workloads (in i7-2600.)

	Execution time	Total insts	LLC refs/sec	LLC misses/sec
Chess	26.0 s	126 B	1.2 M	0.1 M
FaceDetect	38.0 s	203 B	17.9 M	5.3 M
VirusScan	74.5 s	503 B	1.0 M	0.6 M
FeatureDetect	41.7 s	222 B	13.9 M	4.8 M
Bellmanford	147.7 s	508 B	24.1 M	6.0 M

bution. We modeled both 3G and Wi-Fi networks, but used Wi-Fi environments to focus more on the sever-side performance for evaluation. CMcloud can be equally applied to 3G network as well.

Workloads. We implemented five real-world mobile applications listed in Table 3.2. We carefully selected these workloads for the reasonable execution latency, while they contend for the shared resources (e.g., last-level cache.) Chess calculates the latency for a computer to find the next move, FaceDetect and FeatureDetect identify human faces and various features from a given image, VirusScan compares 1K virus signatures with 1GB of cloud data, and Bellmanford finds a shortest path based on a NY-city map. To support offloading, we modified these workloads as proposed in [2]. We also applied Native Interface (JNI) to evaluate memory-intensive workloads.

Clients. We modeled clients as an inflow of offloading requests based on Poisson distribution with 30 requests per minute for the 16-socket cloud. To finish 30 applications per minute, we configured the application ratio as Chess (19.2%), VirusScan (9.7%), FaceDetect (32.3%), FeatureDetect (32.3%), and Bellmanford (6.5%.)



(a) Idle server

(b) Busy server running the five background jobs

Figure 3.7 Accuracy of the performance prediction.

3.4.1 Estimating Target CPU Performance

We first evaluate the accuracy of the proposed performance estimation method using idle target servers by with the reference-model profiling described in Section 3.3.2. Figure 3.7(a) compares the estimation accuracy between the real performance obtained on the target server and the estimated performance of CMcloud. The x-axis indicates three target-server runs for six workloads, whereas the y-axis shows the performance normalized to the reference machine as shown in Table 3.1. Real bar indicates the actual post-offload performance, while CMcloud bar indicates the predicted performance. The results indicate that CMcloud predicts the performance of idle target servers with the average error of only 2.9%. Freq bar indicates the performance only when the CPU frequency is considered for the estimation, which leads to the average error of 10.3%.

Next, we repeat the same experiments when each target server runs a group of five baseline applications in background. Figure 3.7(b) indicates that CMcloud's performance estimation is also accurate even for the highly utilized target servers. The results indicate that CMcloud predicts the performance of busy target servers with the average error of only 5.3%, compared to the 13.4% error of the frequency-only estimation.

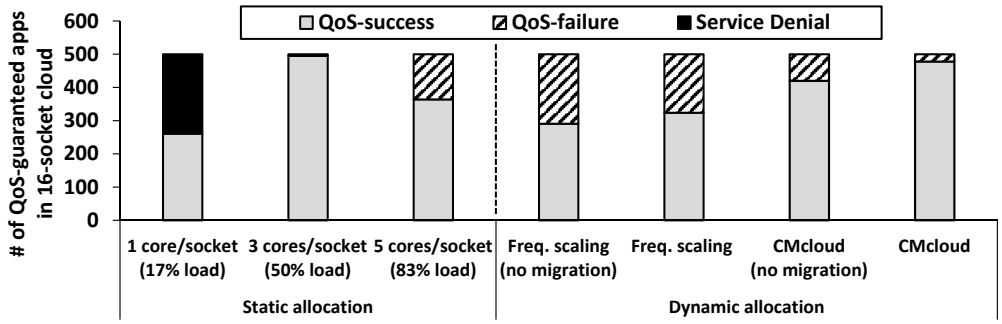


Figure 3.8 Datacenter throughput (out of 500 requests.)

3.4.2 Cost Effectiveness with QoS Requirements

This section evaluates the cost effectiveness of CMcloud by analyzing the improved server throughput and reduced server costs.

Improved server throughput. Figure 3.8 compares the performance and costs of CMcloud against conventional static server allocation schemes. The X-axis lists seven target server allocation schemes: three static allocation schemes and four dynamic allocation schemes including CMcloud. For static allocation schemes, we configured the cloud provider to assign only one application to each socket (17% load,) three applications to each socket (50% load), and five applications to each socket (83% load.) For dynamic allocations schemes, we evaluated a frequency-only estimation model and CMcloud with/without intra-server migration capability. The Y-axis shows, among 500 offload requests, the number of requests successfully completed within the user-agreed deadline (QoS-success) for the entire cloud, the number of requests violating the deadline (QoS-failure,) and the number of requests turned down by the cloud due to insufficient servers.

Among the static allocation schemes, the 17% load scheme shows the lowest per-socket throughput by utilizing only one core per 6-core socket. The 17% load rejects almost half of the requests due to insufficient servers. On the other

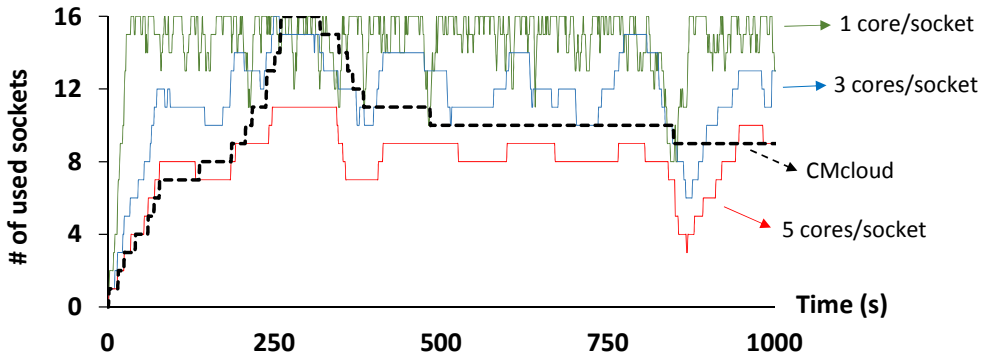


Figure 3.9 Datacenter utilization (out of 16 sockets.)

hand, the 83% load scheme achieves 75% server throughput, while 25% of workloads fail to complete within the deadline. Even though the 50% load shows the maximum throughput in return of 50% server efficiency, this sweet spot will change for different workloads. Therefore, considering the server underutilization and the QoS failure are unacceptable for the cloud business, the static allocations cannot be applied as a cost-effective offload scheme.

Among dynamic allocations, CMcloud achieves almost the ideal throughput and even CMcloud without migration capability outperforms two frequency-only estimation models. The result shows that CMcloud improves the server throughput by 84% over the 17% load scheme. Compared to the 83% load scheme, CMcloud reduces the number of service failures by 83%, while even improving the throughput by 31%. The results also show that both the performance modeling and inter-server migration of CMcloud contributed to the improved server throughput separately.

Reduced server costs. Figure 3.9 shows the number of sockets running applications for the first 1000 seconds. In this experiment, we evaluate the server costs of CMcloud against three static load schemes. As expected, the higher-load allocation policies utilize a smaller number of sockets than lighter-load

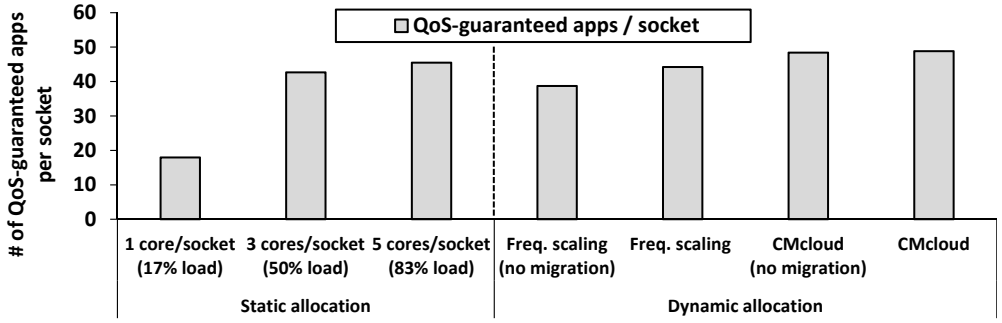


Figure 3.10 Per-socket cost effectiveness.

allocation policies. However, CMcloud only activates the minimum number of sockets by maximizing the throughput, as long as the QoS of applications is not violated. Considering the CMcloud's high throughput shown in Figure 3.8, it is clearly shown that CMcloud consistently operates at lower costs than the 13% and 50% static allocation schemes.

Cost effectiveness. Considering the improved throughput and reduced server costs of CMcloud, Figure 3.10 compares the cost effectiveness of CMcloud against the static allocation schemes. In this figure, we measure the cost effectiveness of the number of applications successfully completed within the deadline per socket, which indicates each socket's cost effectiveness. The results show that CMcloud outperforms all schemes significantly. CMcloud provides a 2.7x higher per-socket throughput over a static light-load scheme (i.e., 17% load.) It should be noted that the relatively high cost-effectiveness of high-load static allocation policy (i.e., 83% load) comes with many QoS failures. On the other hand, CMcloud does not incur unacceptable QoS failures as shown in Figure 3.8.

Table 3.3 Offloading overheads.

	Monitoring overhead	Migration downtime	Profling overhead
Chess	1.58%	45 ms	x150
FaceDetect	0.14%	21 ms	x127
VirusScan	4.15%	14 ms	x180
FeatureDetect	0.71%	12 ms	x124
Bellmanford	3.66%	50 ms	x136

3.4.3 Offloading/migration Overhead

Table 3.3 shows the overhead of performance monitoring, inter-server migration, and reference-model profiling. Both monitoring and profiling overheads are normalized to the execution latency without profiling. The monitoring overhead is small and thus shown in percentage.

Once applications are offloaded to servers, CMcloud must monitor all applications to detect the potential QoS violations and trigger server-to-server migrations to improve the performance. We use KVM's native live migration method, which can migrate an application paying only the minimum performance loss. By modifying the KVM's live-migration source code, we measure the latency from when the VM stops at the source node to when it restarts at the destination node. Table 3.3 shows that both monitoring and migration overheads are minimal.

The static profiling can take a long time as it includes the reuse distance analysis obtained by QEMU emulator. However, it is only a one-time overhead paid by the cloud provider and the overhead is not exposed to users. Moreover, CMcloud requires only a single reference-machine profiling, regardless of its current utilization. It should be noted that a similar kind of static profiling is also required by the existing seminal works[2, 3]. Many proposals to reduce the profiling overhead has been proposed, which is orthogonal to our work.

3.5 Related Work

In this section, we discuss previous work related to CMcloud in the areas of dynamic offloading, performance prediction, performance monitoring, and migration.

Dynamic offloading. MAUI [2] and CloneCloud [3] allow users to execute a mobile application on a cloud. However, these schemes are not suitable for the real-world cloud environment due to the lack of the QoS guarantee of applications and a cost model [33, 34]. ThinkAir [19] proposes an on-demand resource allocation for user-side cost and parallel method execution of a mobile application for the QoS guarantee, but focuses on one automatically parallelizable application instead of simultaneous execution of several applications. In the previous chapter, I focus on reduction in migration overhead by transferring only essential heap objects. In this chapter, Instead, our scheme targets to mobile cloud computing for simultaneous execution of several applications, the QoS guarantee of applications, and minimization of server cost.

Performance prediction. In heterogeneous multi-core systems, PIE [35] and Regression analysis [36] estimate the performance of other cores and assign an appropriate application to an optimal core. These schemes assume that caches have the same size and there is no resource contention. Bubble-Up [37] and Bubble-Flux [38] guarantee QoS of a latency sensitive application. However, the former performs many sensitivity tests with various memory pressures in advance, and the latter does not allow co-location of multiple latency sensitive applications. Mantis [39] can automatically estimate the application performance on various inputs by extracting features related to the performance from an application. For an application with different inputs, we can apply this technique to reduce inaccuracy of profiling.

Performance monitoring. Many researches [40, 41] widely use resource monitoring to detect performance interference. Perf [31] and Oprofile [42] monitor the system resource usage of each application through hardware performance counters. Pin [43] and Valgrind [44] measure what kinds and how many instructions are executed through dynamic instrumentation.

Migrations. Cloud systems migrate VMs to another server for guaranteeing QoS and improve cost effectiveness of clouds. To reduce the downtime of VMs, we adopt Pre-copy [45] as a live migration scheme. We can adopt other live migration schemes [46, 47].

Chapter 4

Application-Specific Execution Offloading for 3D Video Games

In this chapter, we propose our novel offloading approach to enable execution offloading for 3D video games. First, we adopt streaming based techniques into our offloading framework to reduce the data transfer cost of rendering functions. When the rendering functions are being offloaded continuously, the resulting images are streamed to the mobile and only the newly-update application states like the user inputs are transferred to the server. As a result, our framework effectively offloads rendering functions and successfully guarantees quality of service (QoS) of 3D video games in terms of execution time. We also introduce *live offloading*, which allows transferring the current application state before the remote execution actually begins, to make our offloading framework even more effective for better user experience. The manipulated application state during the remote execution is also returned before the remote execution is finished. With live offloading, we can hide the large data transfer cost at the beginning and end of remote execution; it prevents that such a large data cost enlarges

response time and degrades user experience.

4.1 Background and Motivation

4.1.1 Background

The goal of execution offloading is to overcome the lack of computational resources of SMDs by "offloading" the computations of mobile application from SMDs to nearby resource-rich powerful server or cloud. In order to achieve this goal, it dynamically predicts two performance metrics, offloading profit and offloading cost, for each remotely executable method (REM) of a target application. The offloading profit of a method means an expected gain obtained by offloading the method to the remote server, in terms of execution time and energy consumption. The offloading cost of a method is roughly divided by two factors, data transfer cost and runtime cost.

To actually execute a certain method remotely, the current application state should be transferred to the server and the result of the method should be also returned to the SMD as explained in Chapter 1. The data transfer cost is an required cost to transfer the state and result over mobile network. The runtime cost is caused by the offloading process, which is an execution sequence to actually run the method on the server. The offloading process occurs if and only if the method is profitable; which means that the offloading profit of the method is larger than the offloading cost, so the act of offloading the method guarantees the performance enhancement. Based on this strategy, execution offloading effectively improves the performance of mobile applications by making use of resource-rich servers in their proximity.

Figure 4.1 shows an example of the whole offloading process at run time. When a REM of the target application is just called (1), the offloading framework checks the decision for the REM (2). Then the current application state is

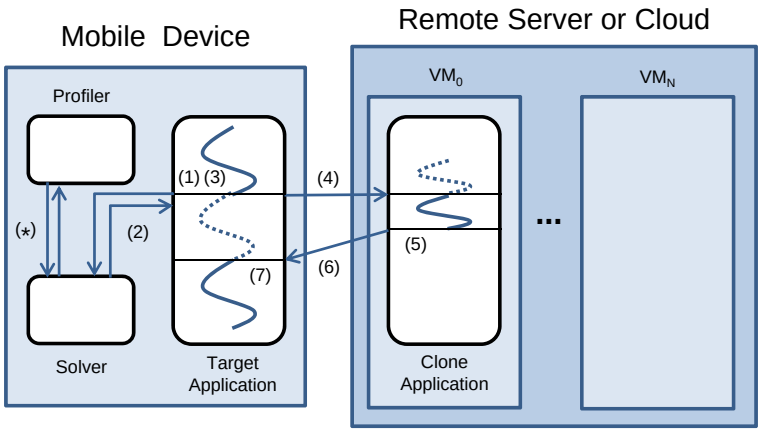


Figure 4.1 An example of the runtime offloading process.

captured and transferred to the remote server (3)(4). After the remote execution is finished, the modified state is also captured and returned to the mobile device (5)(6). Finally, the offloading framework merges the returned state and resumes the target application(7). Separately to this process, the solver periodically generates the decisions for each REM based on the runtime performance factors (*).

4.1.2 Motivation

In execution offloading, the overall performance enhancement depends on how many REM is actually offloaded to the remote server. In other words, the number of profitable REM affects the performance of execution offloading. It is also important that how much gap exists between the offloading profit and cost of each profitable REM. As the offloading profit exceeds the offloading cost much more, execution offloading can also improve the performance of mobile application even more. For maximizing the performance enhancement via execution

offloading, therefore, it is necessary to make each REM profitable as many as possible by reducing their offloading cost. In Chapter 2, I already explained how this goal could be achieved by reduced the data transfer cost of REM based on compiler code analysis.

As shown in prior works [2, 3, 48], the offloading strategy and optimizations about the offloading profit and cost are very effective for usual applications in most cases. For 3D video game[1], unfortunately, we have observed that they do not apply all the time. It is because of rendering functions, which are one of the time-consuming key functions of 3D video game. Because rendering functions generate a stream of images continuously at run time, their offloading cost is quite large. For example, almost 70 megabytes of images are generated in a second where the video game runs at 30 frame rates, 10 bits color depth on 1280 * 720 resolution. In this case, the offloading framework should pay the large data transfer cost to return back the image stream from the server to the mobile, until the remote execution is over. Such a large cost make rendering functions unprofitable, even though those functions could benefit from powerful GPU and other computational resources equipped in the remote server. To fully maximize the performance enhancement by execution offloading on 3D video games, therefore, it is needed to make rendering functions profitable by reducing their large data transfer cost.

[1]The reason why we chose 3D video games as our target application is already explained in Chapter 1.

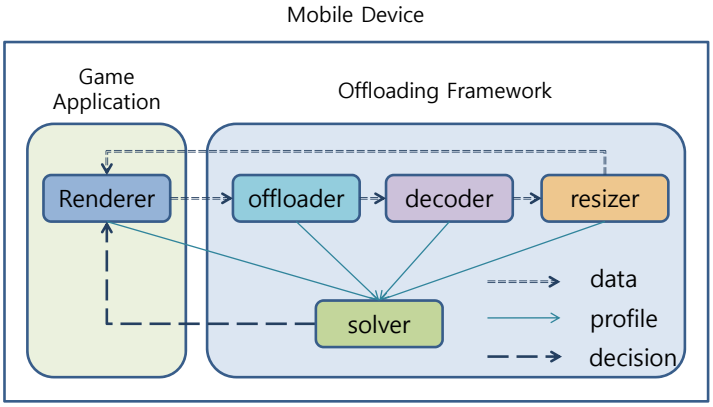


Figure 4.2 A streaming-based offloading framework.

4.2 Application-Specific Execution Offloading

In this section, we explain our offloading framework and optimization techniques that enable execution offloading for 3D video game.

4.2.1 Offloading Framework for Reducing Data Transfer Cost

In execution offloading for 3D video games, the most important design goal is to make rendering functions be profitable. To achieve this goal, we designed a streaming-based offloading framework which successfully reduces the data transfer cost of rendering functions. Figure 4.2 shows our offloading frame and how it works.

Our framework consists of not only basic offloading modules for typical methods but also dedicated modules, such as a decoder and resizer, for rendering related methods. We define rendering related method (RRM) as a method whose output is a complete image. In offloading process, the dedicated modules focus on reducing the data transfer cost of each remotely executable RRM. When the framework decides to offload any RRM based on runtime performance factors,

the application state needed for the RRM is captured and transferred to the server through a communication specific module called offloader. The server returns its output image to the mobile device after the remote execution of the offloaded RRM is finished. At this moment, the output image is compressed to lighten the data transfer cost; the decoder in the mobile device decompress the returned image and pass it to the resizer. The resizer resizes the passed image to fit it into the display resolution of the device. Finally, the renderer in the target application shows the resized image to the user instead of rendering its own image. During the whole offloading process, each module periodically reports its execution time as a performance factor to the solver. The solver uses the factors to make offloading decisions, together with another runtime performance factors like the network latency.

The modules in Figure 4.2 including the dedicated modules for RRMs are implemented as an independent thread, to simultaneously deal with multiple images if the remote execution of RRM is being maintained continuously. Our framework also transmits the newly updated application state only in the case of such a continuous remote execution. When the offloading process capture the application state for RRMs, it compares the state to the latest one and transfers the difference only. Usually, the user input is the difference in 3D video games. By transmitting the user input only, our framework decreases the data transfer cost of RRMs even more as the remote execution continues further[2].

Note that our dedicated modules works only for RRMs. Similar to MAUI approach, typical offloading process is launched for typical methods which does not produce any images. The application developer can distinguish between RRMs and normal methods by wrapping their RRMs with pre-defined method

[2]This strategy has been introduced by MAUI first. Streaming gaming also transmits the user input only at run time.

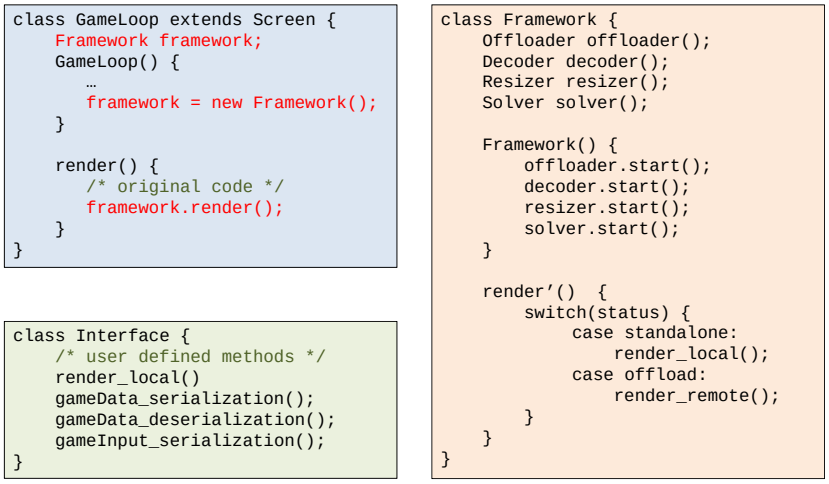


Figure 4.3 Code example for application developer.

signature. Besides, the developer can optimize the offloading process by writing their own serialization and de-serialization method for the application state. Figure 4.3 shows a simple code example to use our framework. Inside the game class `GameLoop` in Figure 4.3, the original rendering method `render` is replaced by the method `render'`, which is a wrapper method of `render`. By calling `render'` instead of `render` and importing the framework class `Framework`, our offloading framework can be easily adopted into the target application. The developer can also implement `Interface` class to optimize the offloading process even more, by writing their own serialization and de-serialization method.

4.2.2 Live Offloading to Guarantee QoS

In the previous subsection, we describe how our offloading framework reduces the data transfer cost of RRMs with streaming-based remote execution. Although these techniques successfully reduces the data transfer cost in the middle of remote execution, unfortunately, large amount of application state is still transferred at the beginning and end of remote execution. In execution offloading, all of application state (or data) resides in the mobile device basically while the data is hosted on the server in streaming gaming. Therefore, the application state needed to run any method should be inevitably transferred to the server before the remote execution starts, as we explained in Section 4.1.1. If the size of such application state is large enough for any remote execution, the runtime overhead to transfer the state fugitively degrades QoS of the remote execution even though the overall offloading cost exceeds the offloading cost.

To tackle this problem, we propose live offloading that allows transmitting the application state needed for remote execution in advance. Instead of transmitting the application state after the local execution is suspended, we simultaneously transfer the state while the local execution is still running. To implement live offloading, we add new execution phases called warm up and cool down to our execution model. Figure 4.4 represents a diagram of the execution phase cycle for target application with live offloading.

When the performance of a target application is near to the 'boundary performance', which is a performance border between local and remote execution, the new execution model changes its execution phase from the stand-alone (or mobile-only) phase to the warm up phase first, instead of starting the remote execution directly. In the warm up phase, the application state is transferred to the server in parallel with the local execution. The actual remote execution

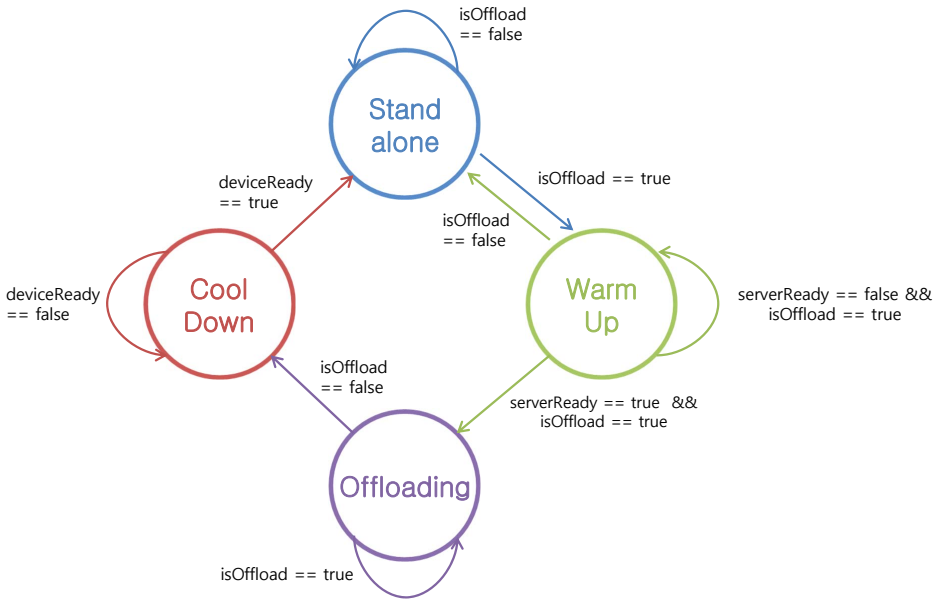


Figure 4.4 An execution phase cycle of live offloading.

can be started only after the transmission of the application state is completed so the remote execution is available. The runtime procedure of the cool down phase is almost similar to the warm up phase; the updated application state is transmitted to the mobile device before the remote execution is finished yet. Finally, the target application runs locally again after the cool down phase is over. One of the concerns of applying live offloading is that the application state updated in the warm up and cool down phase could be missed. To explain this, let assume that the application state at the beginning of the warm up phase is S1 and the state at the end of the phase is S2. The latest state S2 could be differ from the captured state S1, which is also transferred to the server, according to whether the local execution updates its state in the warm up phase or not. If the offloading framework does not adjust the difference between S1 and S2 appropriately, the remote execution may cause semantic inconsistency, which is the case when the result of remote execution is different to mobile-only execution's [48], due to the out-of-date state S1. In 3D video games, especially, such a semantic inconsistency problem leads to a serious degradation of user experience; the users may immediately notice that their inputs are not reflected to the game play, and that may also make them very uncomfortable even if the warm up or cool down phase continues for a just few seconds.

To prevent the semantic inconsistency problems caused inadvertently by live offloading, we store the accepted user inputs into a dedicated queue, called 'input queue', in the warm up phase. After transmitting the captured application state, our offloading framework starts to dequeue an input from the input queue and transfers it to the server, until the the queue is empty. The warm up phase is also over at this moment. Since only a few milliseconds are needed to transfer a single input and the number of the input accepted in a second is limited, fortunately, the duration of the warm up phase is just a few seconds

in most cases. Similarly, the user inputs transferred in the cool down phase are simultaneously stored into the input queue, and updated to the returned application state right after the state is arrived to the mobile device. Based on these techniques and live offloading, we successfully prevent that the large data transfer cost and the unexpected semantic inconsistency problems degrade user experience on our offloading framework.

4.3 Evaluation

We built our framework and optimization techniques based on libGDX [49], a open source multi-platform Java game development framework. Most of our framework is written in traditional Java language and only some computation intensive parts are implemented by using Java Native Interface (JNI). As our benchmark application, we chose one of the 3D demo games provided by libGDX and built two different versions of the benchmark for both Android and Intel x86 architecture. For the smartphone and server, we chose a Galaxy Nexus with dual-core 1.2 Ghz CPU and 1 GB of RAM and a quad-core desktop with a 3.1 GHz CPU and 8 GB of RAM running Ubuntu 11.10.

Figure 4.5 compares the performance result of two game plays of our benchmark with and without execution offloading, in terms of frames per second (FPS). From 100 objects initially, we increased the number of the objects in each play by 100 objects for every five second to drop the performance of the benchmark artificially in this evaluation. As a result, the FPS of the mobile-only play, represented as a dashed blue curve, dropped under 10 FPS in the end. For the play with execution offloading, by contrast, the remote execution was started around 17 second so the FPS of the play was stabilized above 20 FPS as depicted in a red curve. It was possible to achieve such an impressive enhancement by not only simply offloading the RRMs to the server, but also

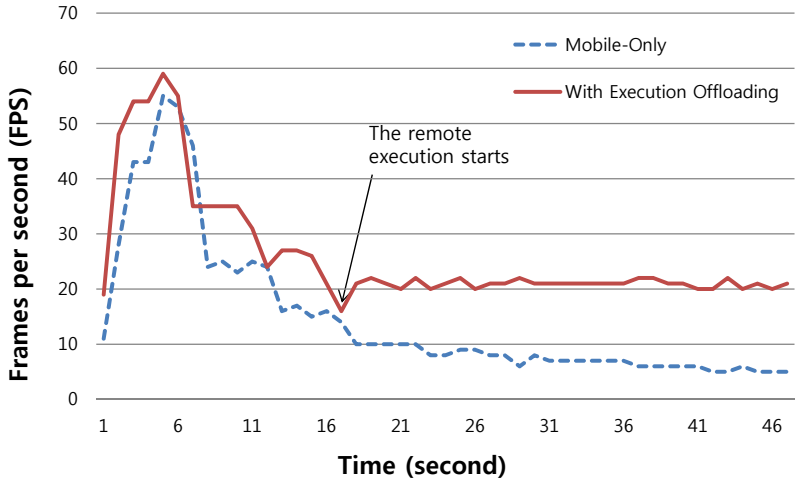


Figure 4.5 The performance result of two game plays with and without execution offloading.

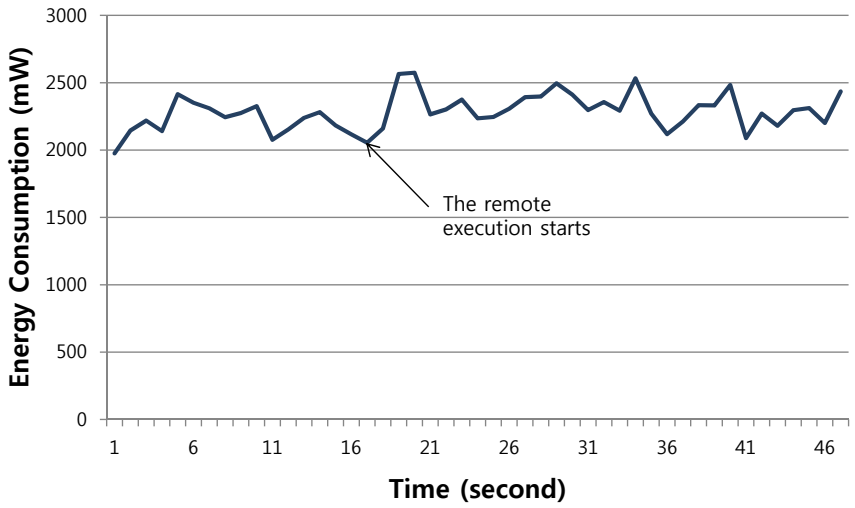


Figure 4.6 The energy consumption of the smartphone with execution offloading.

reducing the data transfer costs of the RRMs successfully as we described in Section 4.2.

In Figure 4.6, the energy consumption of the smartphone for the play with execution offloading in Figure 4.5 are presented. We used an off-board monitor [13] to evaluate the energy consumption. The average energy consumption was increased by 4.5% from 2215.56 mW to 2315.74 mW, after the remote execution was started at 17 second. It seems that the major reason of the additional consumption was the energy overhead to run the dedicated modules and the network equipment for the image streaming. It is also due to the policy of our offloading solver, which concentrates to guarantee the minimum FPS even if more energy is spent; if we gave priority to energy efficiency, our offloading framework should try to save energy first rather than guarantee the minimum FPS. We expect that the video game user may accept the remarkable performance benefit of execution offloading shown in Figure 4.5, regardless of slightly increased energy consumption.

We also measured the impact of live offloading on the response time, which is the time required to get the first image since the remote execution begins. The average response time was almost 1000 ms without live offloading, which means that the user should wait for one second until the remote execution is actually started. However, such a delay was successfully improved by live offloading; the average response time was reduced to near 300 ms, and almost 700 ms of the runtime overhead to transfer the application state was hidden in the warm up phase. This example demonstrates that our live offloading technique can be quite useful to improve user experience on execution offloading.

4.4 Related work

Many researches have been proposed that aim to empower modern mobile devices with surrounding powerful servers. One of the earliest studies was done by Satyanarayanan et al. [7], who have developed versions of *ISR* systems for the past decade. To offload a process running on the device, they migrates the full virtual machine (VM) or OS image along with the process. Not surprisingly, the amount of transferred data for VM migration tends to be huge (around in the order of gigabytes). To lighten the load, they proposed the *dynamic VM synthesis* approach [5] where a small VM overlay is sent by a mobile device to the *cloudlet* (nearby small cloud) that is already installed with the base VM which the overlay was derived from. The overlay size was reported about one order of magnitude smaller than the full VM size, so they claimed that the approach might be feasible for mobile computing using fast wireless LANs like Wi-Fi. However, even that figure would be still too high for 3D video games which require extremely high performance and short delay time. Wishbone gives a solution for optimal partitioning of sensor network application code across sensors and servers. It statically partitions the application code based on profile data that include the computational and network load by using an integer linear program to find the minimum use of CPU and network bandwidth. Wishbone [8] guarantees that the optimal partitioning can be predetermined regardless of the target hardware platform because it only targets a confined area of applications where a division of subtasks is fairly clear. OLIE [18] collects the current status of the memory utilization and available network bandwidth to decide whether offloading should be triggered at run time. But the main goal of OLIE is to overcome only the memory resource constraints of mobile devices. This is deemed relatively simple as compared to optimizing energy consumption and execution

times, which is one of the our goals. Because of their limitations, these studies are not suitable neither for boosting modern 3D video games.

In order to make mobile cloud computing more feasible and adaptable for the mobile device environment, recent execution offloading researches dynamically offload their computation to the remote server, based on various runtime performance factors such as execution time, energy consumption, and network latency. CloneCloud [3] suggests dynamic execution offloading approach by modifying the mobile execution environment, Dalvik VM, to capture the current application state. Because of their approach, CloneCloud do not need to modify the application code. MAUI [2] is a RPC based offloading architecture which decides at runtime which methods should be offloaded based on the best energy savings possible under the current runtime performance factors. MAUI requests special user annotations on the application code to mark REMs. Although the basic architecture of our offloading framework is inspired by MAUI, these studies target general mobile applications while our work focuses on accelerating 3D video games with execution offloading.

More recently, several approaches have been proposed to improve the performance of execution offloading. ThinkAir [19] suggests a dynamic resource allocation scheme, which allocates more than one clone VM for the offloaded application to exploit parallelism and to relieve the lack of memory space. By adopting distributed shared memory (DSM) into its offloading framework, COMET [20] expands the range of remotely executable code and consequently allows multiple threads to be simultaneously offloaded. Kovachev et al. [21] present their middleware which serves more sophisticated profiling, monitoring and offloading decision. Unfortunately, none of them explicitly proposes how to reduce the huge data transfer cost caused by the offloaded RRMs in 3D video games.

Unlike the studies mentioned above, Odessa [6] suggests an execution offloading approach tailored to interactive perception applications such as face, object, pose, and gesture recognition. Odessa identified the performance factors of perception applications based on their elaborate analysis, and the factors make their profiler and offloading solver be lightweight and simple. As a result, Odessa successfully improves the performance of perception application by exploiting parallelism of those applications. However, there are several hurdles to apply Odessa to 3D video games; first of all, the developer who tries to apply Odessa must use the specific development framework. Another hurdle is that the characteristic of parallelism may differ between perception applications and 3D video games. Regardless of such hurdles, however, the methodology proposed by Odessa gives many inspirations to our future research.

Streaming gaming is another well-developed solution for boosting 3D video games with powerful servers. In streaming gaming, most of the core functions of a game application are executed on the gaming server and all of the necessary data to run those functions also resides in the server; the client device only runs user interface (UI), which passes the user input to the server and shows the streamed video to the user. One of the weaknesses of streaming gaming is that its runtime performance heavily depends on the network condition, because the generated game video by the server should be continuously streamed over network until the game is over. In order to enable streaming gaming for mobile devices by getting over the weakness, Wang et al. [50, 51, 52, 53] present an analysis on the performance factors that affect the performance and user experience [50], and a streaming gaming framework which adaptively adjusts the streaming quality based on those factors [51]. In respect that both approach try to empower the resource-constrained mobile device with powerful servers, execution streaming is similar to streaming gaming. It is also true that our work has

been greatly inspired by streaming gaming, especially for our dedicated modules to reduce the data transfer cost of RRMs as we explained in Section 4.2. In spite of that, there is the biggest difference between our execution offloading approach and streaming gaming; in execution offloading, the execution transition from local to remote execution and vice versa is relatively flexible compared to streaming gaming where the task partition between the server and the client is fixed. When the network latency is too low so that the network connection of the mobile device is going to be disconnected, for example, execution offloading can handle this situation by finishing the remote execution and running every REMs locally. Because of its flexible execution model, we expect that execution offloading has its own application area especially in the mobile computing market.

Chapter 5

Conclusions

In this dissertation, we proposed various optimization techniques on execution offloading for more efficient mobile cloud computing. First of all, we proposed the optimization techniques of assisting execution offloading by reducing the size of transferred application state. While the existing work based on the full execution offloading has focused on finding optimal partitions for given computational resources and network conditions, they did not make active effort to reduce the state size which, as we proved, has been a crucial element for the success of execution offloading. We have also demonstrated that careful compiler analysis greatly helped our optimization techniques to effectively achieve our research goal, thereby enhancing the efficiency of mobile computing with the computational support of clouds. The experiments exhibit that the reduced size positively influences not only the transfer time itself but also the overall effectiveness of execution offloading, and ultimately, improves the performance of our mobile cloud computing significantly in terms of execution time and energy consumption.

It is a major challenge to design cost-effective execution offloading scheme which satisfies the runtime condition of the real-world commercial cloud environments. To achieve this goal, we introduced CMcloud, a novel cost-effective mobile cloud platform, which works nicely under the real-world cloud environments. CMcloud reduced the cost of offloading by improving the server utilization significantly, while achieving the user-expected offload performance. Our implementation shows that CMcloud can improve the datacenter throughput by 84% over a conventional static light-load scheme (or a 2.7x higher per-socket throughput.) Alternatively, CMcloud reduces the number of service failures by 83% over a static high-load scheme, while even improving the throughput by 31%. To the best of our knowledge, CMcloud is the first cost-effective mobile cloud platform which allows an oversubscribed offloading without affecting the QoS of mobile applications.

Lastly, we suggested a streaming-based execution offloading framework which enables execution offloading for 3D video games by reducing the large data transfer cost of the rendering related methods. We also introduced live offloading technique which allows transferring the needed application state before the beginning and end of the remote execution, to prevent that the data transfer cost of the state affects the user experience. The experimental results demonstrated that our offloading framework effectively enhances the performance of 3D video game, and live offloading technique successfully reduces the delay time caused by the data transfer cost at the beginning of the remote execution.

Bibliography

[1] Markets and Markets, "World Mobile Applications Market - Advanced Technologies, Global Forecast(2010 - 2015)," 2010. [Online]. Available: http://www.marketsandmarkets.com/

[2] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. ACM MobiSys*, 2010.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proc. ACM EuroSys*, 2011.

[4] R. Ma and C.-L. Wang, "Lightweight application-level task migration for mobile cloud computing," in *Proc. IEEE AINA*, 2012.

[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based Cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14 –23, 2009.

[6] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proc. ACM MobiSys*, 2011.

[7] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D. R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D. J. Farber, M. A. Kozuch, C. J. Helfrich, P. Nath, and H. A. Lagar-Cavilla, "Pervasive Personal Computing in an Internet Suspend/Resume System," *IEEE Internet Computing*, vol. 11, no. 2.

[8] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: profile-based partitioning for sensornet applications," in *Proc. USENIX NSDI*, 2009.

[9] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Proc. ACM/IFIP/USENIX Middleware*, 2009.

[10] Gartner, "Gartner Says Worldwide Video Game Market to Total $93 billion in 2013," 2013. [Online]. Available: http://www.gartner.com/newsroom/id/2614915

[11] Android Developers. Managing your app's memory. [Online]. Available: http://developer.android.com/training/articles/memory.html

[12] S. S. Muchnick, *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[13] Monsoon Solutions Inc. Monsoon power monitor. [Online]. Available: http://www.msoon.com

[14] SciMark 2.0. [Online]. Available: http://math.nist.gov/scimark2

[15] FBReader. [Online]. Available: http://www.fbreader.org

[16] Word frequency data. [Online]. Available: http://www.wordfrequency.info

[17] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Proc. MobiCASE*, 2010.

[18] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Proc. IEEE PerCom*, 2003.

[19] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, 2012.

[20] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proc. ACM OSDI*, 2012.

[21] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Proc. IEEE ISPA*, 2012.

[22] A. Kivity, Y. Kama, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[23] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu, "VMware Distributed Resource Management: Design, Implementation, and Lessons Learned," *VMware Technical Journal*, pp. 45–64, 2012.

[24] P. G. Emma., "Understanding some simple processor-performance limits," in *IBM journal of Research and Development*, May 1997.

[25] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *ASPLOS '06*.

[26] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Interaction cost and shotgun profiling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, no. 3, pp. 272–304, Sep. 2004.

[27] "Intel 64 and ia-32 architectures optimization reference manual," no. 248966-026, April 2012.

[28] Q. Liang, "Performance monitor counter data analysis using counter analyzer," in *IBM developerWorks*, Feb 2009.

[29] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI '03*.

[30] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05, 2005.

[31] A. C. de Melo, "The new linux 'perf' tools," Slides from Linux Kongress, http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf, 2005.

[32] "Android-x86," http://www.android-x86.org.

[33] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 2011.

[34] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, Jan. 2013.

[35] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)," in *ISCA '12*.

[36] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite Cores: Pushing Heterogeneity into a Core," in *MICRO '12*.

[37] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations Categories and Subject Descriptors," in *MICRO '11*.

[38] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ISCA '13*.

[39] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic Performance Prediction for Smartphone Applications," in *ATC' 13*.

[40] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *CGO '10*.

[41] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An Analysis of Performance Interference Effects in Virtual Environments," in *ISPASS '07*.

[42] J. Levon and P. Elie, "Oprofile: A system profiler for linux," http://oprofile.sf.net, 2004.

[43] C.-k. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, S. Wallace, V. Janapa, and G. Lowney, "Pin: Building Customized Program Analysis Tools," in *PLDI '05*.

[44] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI '07*.

[45] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI '05*.

[46] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *VEE '09*.

[47] J. Kim, D. Chae, J. Kim, and J. Kim, "Guide-copy: Fast and silent migration of virtual machine for datacenters," in *SC '13*.

[48] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing," *Mobile Computing, IEEE Transactions on*, vol. 13, no. 11, pp. 2648–2660, Nov 2014.

[49] libGDX, http://libgdx.badlogicgames.com/, 2005, product page.

[50] S. Wang and S. Dey, "Modeling and characterizing user experience in a cloud server based mobile gaming approach," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, Nov 2009, pp. 1–7.

[51] ——, "Rendering adaptation to address communication and computation constraints in cloud mobile gaming," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, Dec 2010, pp. 1–6.

[52] ——, "Cloud mobile gaming: Modeling and measuring user experience in mobile wireless networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 16, no. 1, pp. 10–21, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2331675.2331679

[53] ——, "Adaptive mobile cloud computing to enable rich mobile multimedia applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 4, pp. 870–883, June 2013.

초록

스마트폰 및 타블렛과 같은 지능형 모바일 기기(SMD)는 전체 컴퓨팅 기기 시장에서 사용자들이 가장 선호하는 기기로 빠르게 성장하고 있다. 이러한 시대의 흐름에 따라 사용자들은 모바일 기기에서 비전, 그래픽, 증강 현실과 같이 고도로 복잡한 응용들을 이용하는 데에 보다 더 많은 시간을 할애하고 있다. 그럼에도 불구하고, 제한된 배터리 용량과 느린 네트워크 속도와 같은 주요 자원의 제약으로 인해 모바일 기기에서 이러한 복잡한 응용들을 구동하는 데에는 여전히 한계가 있다. 최근의 연구들은 보다 더 많은 자원을 가진 강력한 클라우드에 모바일 기기를 연결하는 모바일 클라우드 컴퓨팅 기술을 제안하여 이 문제를 해결하고자 하였다. 이러한 시도들은 모바일 클라우드 컴퓨팅을 실제 모바일 컴퓨팅 환경에 효과적으로 적용하기 위한 기술인 실행 오프로딩(혹은 연산 오프로딩) 기술로 이어졌다.

실행 오프로딩의 주요 목적은 프로세스 혹은 메서드를 원격으로 실행함으로써, 모바일 기기의 연산 부담을 강력한 서버 혹은 클라우드로 전가하는 것이다. 이를 위해서는 실행 중의 어플리케이션 상태 정보를 저장하여 네트워크를 통해 서버로 전송하는 과정이 필요하다. 쉽게 예상할 수 있듯이, 이러한 상태 정보를 전송하기 위한 상태 전송 비용은 실행 오프로딩의 성공을 가름하는 핵심 요소이다. 어플리케이션 상태 정보의 크기는 때에 따라 수 메가바이트에 달할 수 있기 때문에, 전송되는 상태 정보의 크기를 줄이는 것은 오프로딩으로 인한 성능 향상을 극대화 하는 데에 있어 매우 중요하다. 본 논문에서는 컴파일러 코드 분석에 기반하여 서버에서 실제로 참조되는 상태 정보만을 전송함으로써 상태 전송 비용을 효과적으로 줄일 수 있는 기술을 제안한다.

실행 오프로딩에 관한 초기 연구들은 여러가지 이상적인 실행 조건들을 가정하였다. 예를 들어, 이들 연구들은 오프로딩의 대상이 되는 서버의 성능이 항상 안정적이며 일정할 것이라고 가정하였다. 하지만 이러한 가정들은 실제 상용화

된 클라우드 환경에서 성립하지 않는다. 이는 클라우드를 구성하는 각각의 단일 서버에 최대한 많은 수의 응용을 실행하는 등의, 수익을 최대화하기 위한 상용 클라우드 환경에서의 실행 특성에 기인한다. 따라서 실제 상용화된 클라우드 환경에 적용하기 위한 보다 더 현실적인 오프로딩 기술을 설계하기 위해서는, 이러한 비용 특화적인 클라우드 플랫폼의 실행 특성을 고려해야 할 필요가 있다. 본 논문에서는 클라우드의 처리량을 최대로 늘릴 뿐만 아니라 오프로딩 된 대상 응용의 성능 요구 조건 또한 만족시킬 수 있는 비용 특화적인 실행 오프로딩 프레임워크를 제안하여 이 문제를 해결하고자 하였다.

실행 오프로딩에 있어서의 또 다른 도전 중의 하나는 응용에 특화된 오프로딩 기술을 설계하는 것이다. 많은 모바일 응용들은 그들 각자의 고유한 특성들을 가지고 있으며, 이러한 특성들 중 일부는 기존 오프로딩 연구들이 세워왔던 전략을 무너뜨릴 수도 있다. 이 때문에 실행 오프로딩을 통해 대상 응용의 성능을 더욱 향상시키기 위해서는 응용의 고유한 특성에 특화된 최적화 기법들을 적용하는 것이 필요하다. 이러한 목표를 달성하기 위한 가능성을 보여주기 위해 본 논문에서는 3D 비디오 게임의 서비스 만족도(QoS)를 효과적으로 보장할 수 있는 스트리밍 기반 실행 오프로딩 프레임워크를 제안한다. 또한 대상 응용에 대한 원격 실행히 실제로 시작되기 이전부터 필요한 응용 상태 정보를 미리 서버로 전송함으로써, 보다 나은 사용자 경험을 제공할 수 있는 실시간 오프로딩 기법 또안 제시한다.

주요어: 모바일 클라우드 컴퓨팅, 실행 오프로딩, 코드 분석, 클라우드 환경, 응용 특화 최적화, 3D 비디오 게임

학번: 2008-20913