Computational Offloading Mechanism for Native and Android Runtime based Mobile Applications

Abdullah Yousafzai[a,*], Abdullah Gani[a,**], Rafidah Md Noor[a], Anjum Naveed[a], Raja Wasim Ahmad[a], Victor Chang[b]

[a]*Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, Malaysia*
[b]*International Business School Suzhou, Xi'an Jiaotong Liverppol University, Suzhou, China*
[c]*School of Computer Information Engineering, Sangji University, Korea*

Abstract

Mobile cloud computing is a promising approach to augment the computational capabilities of mobile devices for emerging resource-hungry mobile applications. Android-based smartphones have opened real-world venues for mobile cloud applications mainly because of the open source nature of Android. Computational offloading mechanism enables the augmentation of smartphone capabilities. The problem is majority of existing computational offloading solutions for Android-based smartphones heavily depends on Dalvik VM (an application-level VM). Apart from being a discontinued product, Dalvik VM consumes extra time and energy because of the just-in-time (JIT) compilation of bytecode into machine instructions. With regard to this problem, Google has introduced Android Runtime (ART) featuring ahead-of-time (AHOT) compilation to native instructions in place of Dalvik VM. However, current state-of-the-art offloading solutions do not consider AHOT compilations to native binaries in the ART environment. To address the issue in offloading ART-based mobile applications, we propose a computational offloading framework. The proposed framework requires infras-

*Corresponding author
**Principal Corresponding author

Email addresses: `yousafzai.abdullah@gmail.com` (Abdullah Yousafzai), `abdullah@um.edu.my` (Abdullah Gani), `rafidah@um.edu.my` (Rafidah Md Noor), `anjum@um.edu.my` (Anjum Naveed), `wasimraja@siswa.um.edu.my` (Raja Wasim Ahmad), `ic.victor.chang@gmail.com` (Victor Chang)

tructural support from cloud data centers to provide offloading as a service for heterogeneous mobile devices. Numerical results from proof-of-concept implementation revealed that the proposed framework improves the execution time of the experimental application by 76% and reduces its energy consumption by 70%.

Keywords: Mobile Cloud Computing, Computational Offloading, Android Runtime Environment, Application Partitioning, Mobile Cloud Scheduling

1. Introduction

Smartphones are gaining immense popularity since the emergence of new mobile applications (e.g., face recognition, natural language processing, interactive gaming, and augmented reality)[1, 2]. These mobile applications are typically resource hungry; they require intensive computation, and high energy consumption [3]. However, the physical size of mobile devices limits their computational resources and battery. As a result, a tension exists between resource-hungry applications and resource-constrained mobile devices; such tension poses a significant challenge for future mobile platform development [4, 5]. To address this challenge, mobile cloud computing (MCC) has been introduced as a promising approach that offloads the extensive computation via wireless access (e.g., cell network, Wifi, or Bluetooth) to resource-rich cloud infrastructure. In MCC, the ecosystem of computational offloading contains different types of computational resources, which may be used depending on their availability and scheduling decisions.

Figure 1 presents a generic view of an MCC environment. The foremost available resource used by mobile applications is the cloud service providers (CSP); they provide on-demand services (i.e., software, platform, and infrastructure) in a pay-as-you-go system [6]. The second type of resource is the cloudlet [7], a nearby (local) Internet-enabled rich computing infrastructure that is connected to mobile devices using wireless access. The third type of resource is the local proximate mobile cloud [8], which is based on the formation of an ad-hoc network of devices within vicinity to collectively serve one another using either Wifi or Bluetooth network interfaces.

For a mobile device, the simplest means of augmentation is using mobile cloud through service-oriented or client/server patterns. Such augmentation is feasible assuming that the requested service, application source code, or application binaries are available on server. Despite its simplicity, this approach

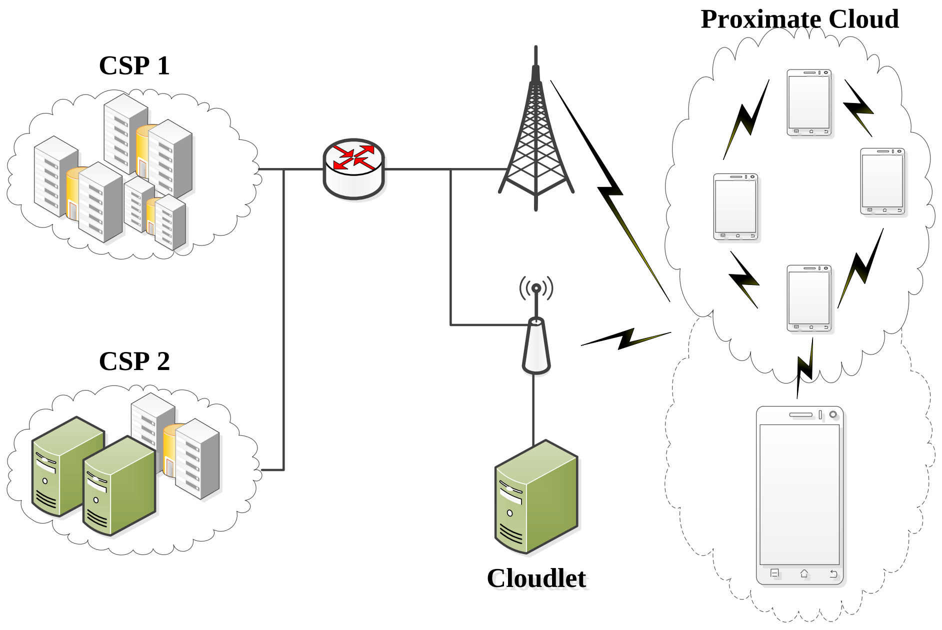


Figure 1: A Generic View of Mobile Cloud Environment

may result in computational losses and waiting times in case of network disconnection or service disruption. The most common computational offloading mechanism is code migration, which migrates intermediate-level instructions between a mobile device and a server. These intermediate-level instructions must be executed on the same type of application-level VM (ALVM) (i.e., Dalvik VM) on both the mobile device and server. The literature also reveals offloading mechanisms that consider thread-state migration or thread-state synchronization ALVM. Both code migration and thread-state synchronization are highly dependent on ALVMs. That is, the computational offloading mechanisms of Android-based smartphones depend on Dalvik VM (DVM). This dependency invalidates the mechanisms for the newly launched Android Runtime (ART) environment. ART has obvious benefits in terms of execution time and battery consumption. ART uses ahead of time (AHOT) compilation to transform device-independent DEX code into device-specific native code [9].

The mobile cloud approach significantly augments the computational capabilities of mobile devices [10, 11]. However, the development of a unified and comprehensive MCC system remains as a challenging task. The difficulty lies in the heterogeneity of hardware and software platforms available on mobile devices and the cloud system. To overcome this above-mentioned challenges, we propose a unified computational offloading framework for native and ART-based mobile applications. The proposed framework will not transfer an application binary or its source code from the mobile device to the server. The application must be modified by developers to support the offloading primitives required by our proposed framework.

The rest of the article is organized in five sections. Section 2 presents an overview of existing computational offloading mechanism through a classification. Section 3, discusses the arguments on why a new process based migration is required. Section 4 presents the details about the structural and functional components of our proposed framework. Section 5 outlines a proof-of-concept experiment, along with the limitation and future prospects of the proposed framework. Finally, Section 6 concludes the article.

2. Related Work

Previous experimentations confirm that remote execution can potentially reduce the power consumption and execution time of applications running on weak smartphones [12]. In a mobile cloud environment (MCE), most of the existing computational offloading mechanisms, which improves the performance and the battery consumption of mobile devices, can be classified under three broad categories, namely, i) VM/phone clone migration, ii) code migration, and iii) thread-state migration.

2.1. VM/Phone Clone Migration

VM/phone clone migration uses virtualization technology to maintain a synchronized mirror for each connected smartphone on a computing infrastructure. This mechanism allows certain operations to be performed directly on the mirror. A high-level diagram of components and mechanisms involved in VM-based augmentation of mobile devices is presented in Figure 2.

Research on Cloudlets [7] , paranoid Android [13] , virtual smartphone [14], and phone mirroring [15] are prominent in the literature on this class of computational offloading. Cloudlets[7] proposes the use of local resource-rich computers, to which a smartphone connects over a single-hop wireless LAN

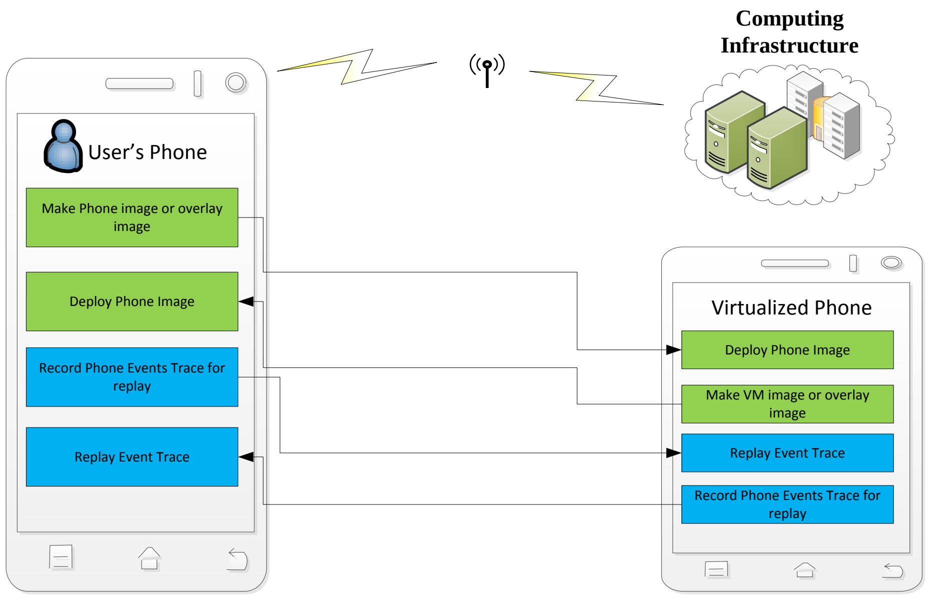
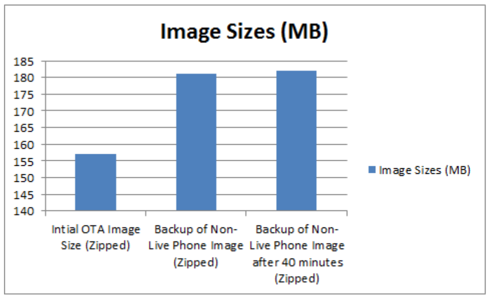
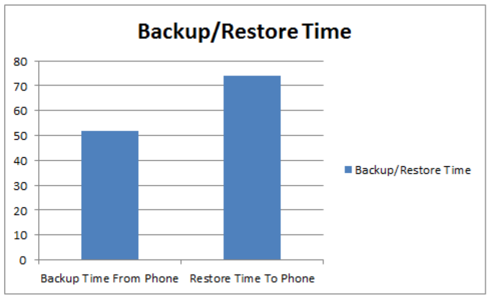

Computing Infrastructure
User's Phone
Make Phone image or overlay image
Deploy Phone Image
Record Phone Events Trace for replay
Replay Event Trace
Virtualized Phone
Deploy Phone Image
Make VM image or overlay image
Replay Event Trace
Record Phone Events Trace for replay

Figure 2: VM Based Augmentation of Mobile Devices

and migrates a current system image. Paranoid Android [13] uses QEMU for running replica Android images in the cloud to enable multiple exploit and attack detection techniques to run simultaneously with minimal effect on phone performance and battery life. Virtual Smartphone [14] uses Android x86 port to efficiently execute Android images in the cloud on VMware ESXi virtualization platform. Phone mirroring [15] framework maintains a synchronized mirror for each connected smartphone on a computing infrastructure, which allows certians operations to be performed directly on the mirror. Cloud-based augmentations of smartphones using VMs/phone clones either deliver or obtain an overlay image (i.e., the difference between two consecutive VM images) or a replay trace to and from the computing infrastructure (Figure 2. Through the overlay image, smartphones synchronize part of execution on the remote VMs and vice versa. However, this mechanism is hindered by the amount of data transfer [16]. An example of the image size (in MB) for a non-live image transfer of An android Jelly-Beans phone is presented in Figure 3 (a). The time required to backup and restore these images from and to the phone is illustrated in Figure 3 (b). To reduce the communication overhead caused by large file sizes, replay mechanisms [16–19], in which execution of mobile device instructions are captured as a trace and then executed on remote VMs and vice versa, can be used while synchronizing remote VMs with phone state. However, reply mechanisms generate large trace files and require extensive modifications to the virtual machine monitors (VMM). From a practical point of view, the phone clones/VMs needs the same hardware platform as the server side to retain a working synchronized image.



(a)



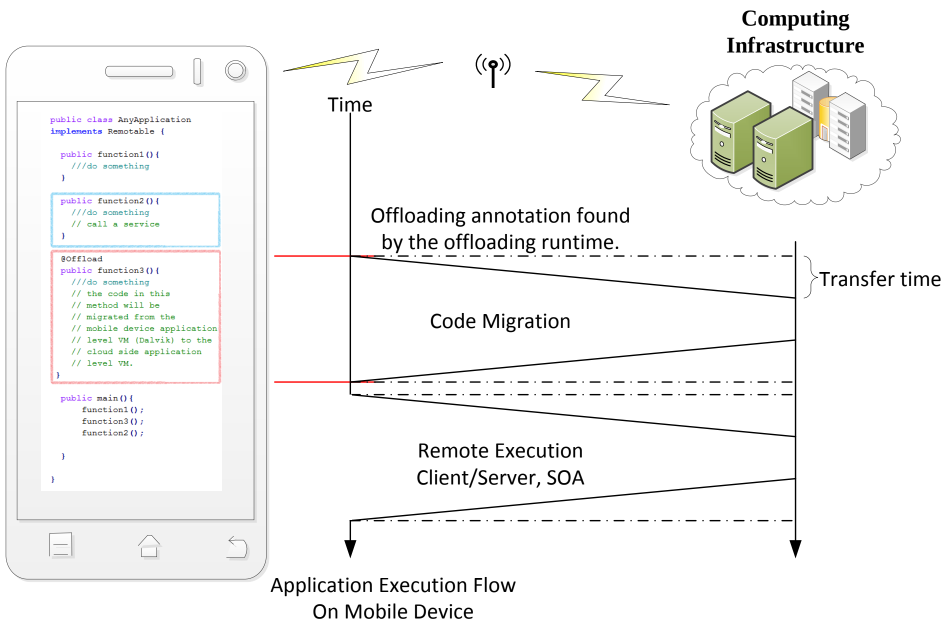
(b)

Figure 3: (a) Phone Image Sizes (in MBs) (b) Phone Image Backup and Restore Time (in seconds)

2.2. Code Migration and Delegation

The most popular and common technique to leverage the computational power of cloud in mobile devices is code migration and delegation. This technique involves the delegation of code execution to remote cloud servers [5] by either migrating a platform-independent intermediate code or using a service-oriented client/server setup [20]. Code migration and remote execution are illustrated in Figure 4. The difference between code migration and remote execution via plain client/server or SOA fashion is clearly presented in the figure. When a server is disconnected or unavailable the mobile application can be locally executed on a mobile device in code migration. By contrast, the SOA-based execution halts until the server becomes available.

Many attempts have been done in past [21–29] to enable remote execution using code migration. Remote execution using code migration improve the performance and battery consumption of mobile devices. Most of these code migration based offloading attempts rely on programmers to specify program partitions using code annotation and skeletons. These program partitions are


Computing Infrastructure
Time
Offloading annotation found by the offloading runtime.
Transfer time
Code Migration
Remote Execution Client/Server, SOA
Application Execution Flow On Mobile Device

```
public class AnyApplication
implements Remotable {

  public function1(){
    ///do something
  }

  public function2(){
    ///do something
    // call a service
  }

  @Offload
  public function3(){
    ///do something
    // the code in this
    // method will be
    // migrated from the
    // mobile device application
    // level VM (Dalvik) to the
    // cloud side application
    // level VM.
 }

  public main(){
      function1();
      function3();
      function2();


  }

}
```

Figure 4: Remote Execution using Code Migration and Delegation

utilized for specification of local and remote application execution decisions. A number of the code offloading-based mechanisms [30] replicate the application binary, intermediate representation, or source code at the server and perform remote execution in a client/server or service-oriented manner. Figure 5 provides the result of an experiment performed with basic code migration using two different settings for a 1000 x 1000 matrix multiplication. The figure illustrates the effect of remote execution using code migration on the execution time of an application.

Despite the significant improvement in execution time, code migration does not support existing applications that do not consider a MCE notion. In addition, static partitioning for high-end mobile devices with intermittent connectivity might not be useful. Further, in case of SOA or client/server setup disconnection or server failure, computation may be lost and the execution on the phone may have to be restarted. Finally, if ALVM is not available, code migration is not a trivial task because of the underlying heterogeneity of hardware and software platform across the connected systems.

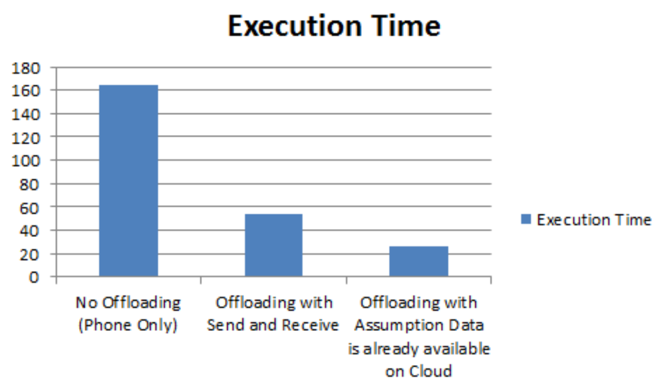


Figure 5: Impact of remote execution using code migration on the execution time (in seconds) of an application.

2.3. Thread-State Migration

In a MCE, thread migration is the migration of low-level thread states (heap contents, stack, descriptors, register values) from one ALVM (i.e. Dalvik, JVM, .Net Runtime) to another.A generalized illustration of thread-state migration is presented in Figure 6.

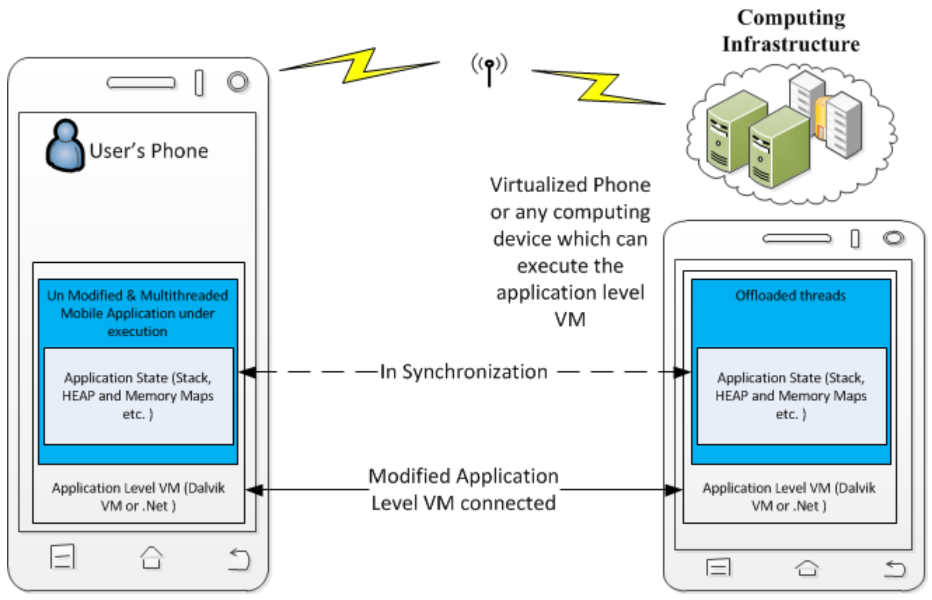

Computing Infrastructure
User's Phone
Virtualized Phone or any computing device which can execute the application level VM
Un Modified & Multithreaded Mobile Application under execution
Application State (Stack, HEAP and Memory Maps etc.)
Application Level VM (Dalvik VM or .Net)
In Synchronization
Modified Application Level VM connected
Offloaded threads
Application State (Stack, HEAP and Memory Maps etc.)
Application Level VM (Dalvik VM or .Net)

Figure 6: Generalized representation of thread synchronization used in MCE

Thread-state migration mechanism is strictly dependent on ALVMs as these ALVMs provides the abstraction and interoperability of threads across different hardware platforms. These ALVMs require massive modifications to enable thread-state synchronization mechanisms. CloneCloud[10] and its variants [31] as well as COMET[32] exploit the concept of thread migration to improve the overall performance of mobile devices. CloneCloud[10] uses a combination of static analysis and dynamic profiling to partition applications at runtime. The application partition is migrated as a thread from a mobile device at a chosen point to the clone in the cloud. The partition is executed in the cloud for the remainder of the process. Then, the migrated thread is reintegrated back to the application executing on a mobile device. On contrary, COMET[32] leverages the underlying memory model of Dalvik runtime and modifies the Dalvik running on a phone and server simultaneously to implement distributed shared memory (DSM). In doing so, thread synchronization is enabled between the mobile device and the server. The modification of Dalvik on smartphone severely affect an application, running locally on the mobile device because the application is not executed, or cannot be offloaded, or the server is disconnected. This phenomenon is tested by executing standard benchmark several times on two similar devices with the same specifications in terms of hardware and software configurations. However, One of the phones has a modified DVM from COMET, whereas the other has the original stock DVM. The benchmark results in Figure 7 measures the mega floating-point operations per second (MFLOPs), which clearly demonstrate the effect on performance caused by the modification of DVM. These thread-state migration mechanisms become automatically invalid in ART environment [33], which is introduced by Google in place of the Dalvik runtime environment. ART features AHOT compilation to device-specific native binaries; this feature expedites the application execution by reducing the overhead caused mainly by the just-in-time (JIT) compilation in DVM [34].

3. Motivation for Process Migration Based Computational Offloading Framework

The classification discussed in Section2 is summarized in Table 1. The table also outlines the dependencies and limitations in current state-of-the-art offloading mechanisms.

The categorized computational offloading mechanisms used in a MCE

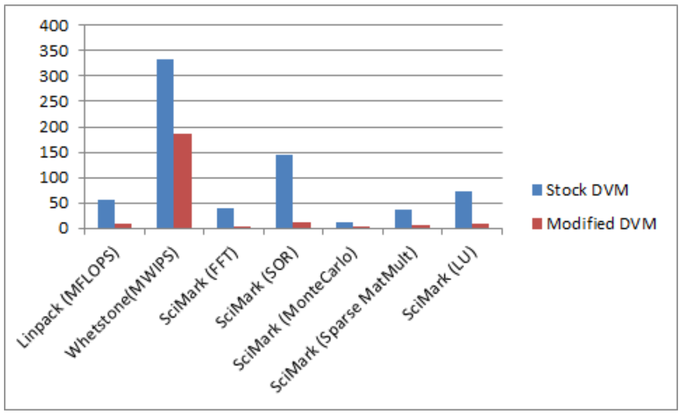


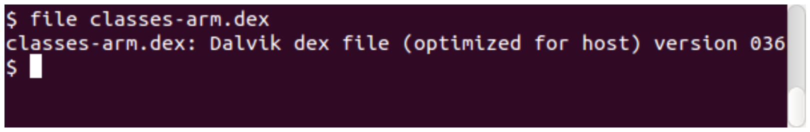
Figure 7: Performance Comparison of Modified DVM vs. Stock DVM using Standard Benchmarks. (in Mega FLOPS.)

Table 1: Comparison of Computational Offloading Mechanisms used in Mobile Cloud Environments

Computational Offloading Mechanisms	Application Modification	Dependency	Major Drawback
VM/Phone Clone Migration	No	Hardware Virtualization	High communication overhead in addition to the overhead of time elapsed in generating overlay state for VM synchrnoization.
Code Migration and Delegation	Yes	ALVMs	Static partitioning using annotations for high-end mobile devices with intermittent connectivity is not useful. Computation is lost and restarted in case of SOA, client/server disconnection, or server failure.
Thread Synchronization	No/ Auto	ALVMs	The overhead (profiling and analyzing or setup overhead) caused by the modification of ALVM to execute an application that does not want/need to be offloaded.

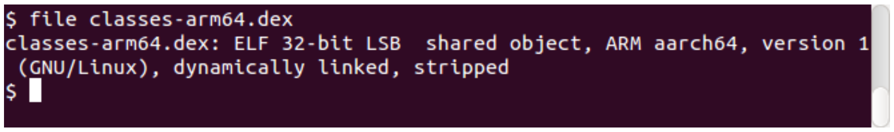
strictly depend on virtualization technology. The mechanisms dependent on application-level virtualization exploit the intermediate code, which can be seamlessly migrated between devices with different hardware and software architecture. By contrast, the VM migration-based computational offloading mechanisms depend on hardware virtualization techniques. ART compiles mobile applications into native machine-dependent binaries upon installation. However, current state-of-the-art offloading solutions do not consider the native code of ART-based mobile applications.

To verify the native code behavior of ART, we gathered the DEX files of an installed application from two Android devices. One device is running Dalvik, whereas the other is running ART. The gathered files are checked with the file[1] tool available in most Linux distributions (Figures 8 and 9). The DEX files gathered from Dalvik- and ART-based phones are in bytecode format and Executable and Linking Format (ELF), respectively.



```
$ file classes-arm.dex
classes-arm.dex: Dalvik dex file (optimized for host) version 036
$ 
```

Figure 8: File type of Dalvik DEX file.



```
$ file classes-arm64.dex
classes-arm64.dex: ELF 32-bit LSB  shared object, ARM aarch64, version 1
 (GNU/Linux), dynamically linked, stripped
$ 
```

Figure 9: File type of ART DEX file.

The ELF file is platform dependent [35]. For instance, the ELF file compiled for an ARM platform cannot be executed on an x86-based platform. Most current state-of-the-art computational offloading mechanisms are exploiting Dalvik bytecode, which is platform independent. The same Dalvik bytecode can be executed without any modification on any platform where DVM is running.

Recently, 1,208,476 mobile applications from the Google Play Store have been statistically analyzed to investigate the number of mobile applications utilizing native libraries [36]. A total of 446,562 mobile applications (37.0%) used at least one natively compiled library. Considering the result of this recent study, computational offloading mechanisms should be reevaluated to overcome the issue of native code. Therefore, we have proposed a process migration-based computational offloading framework.

4. Proposed Framework

Figure 10 illustrates the high-level components of the proposed framework. The framework is modularized into components that are present either at a mobile device or in a cloud server.

[1]http://linux.die.net/man/1/file

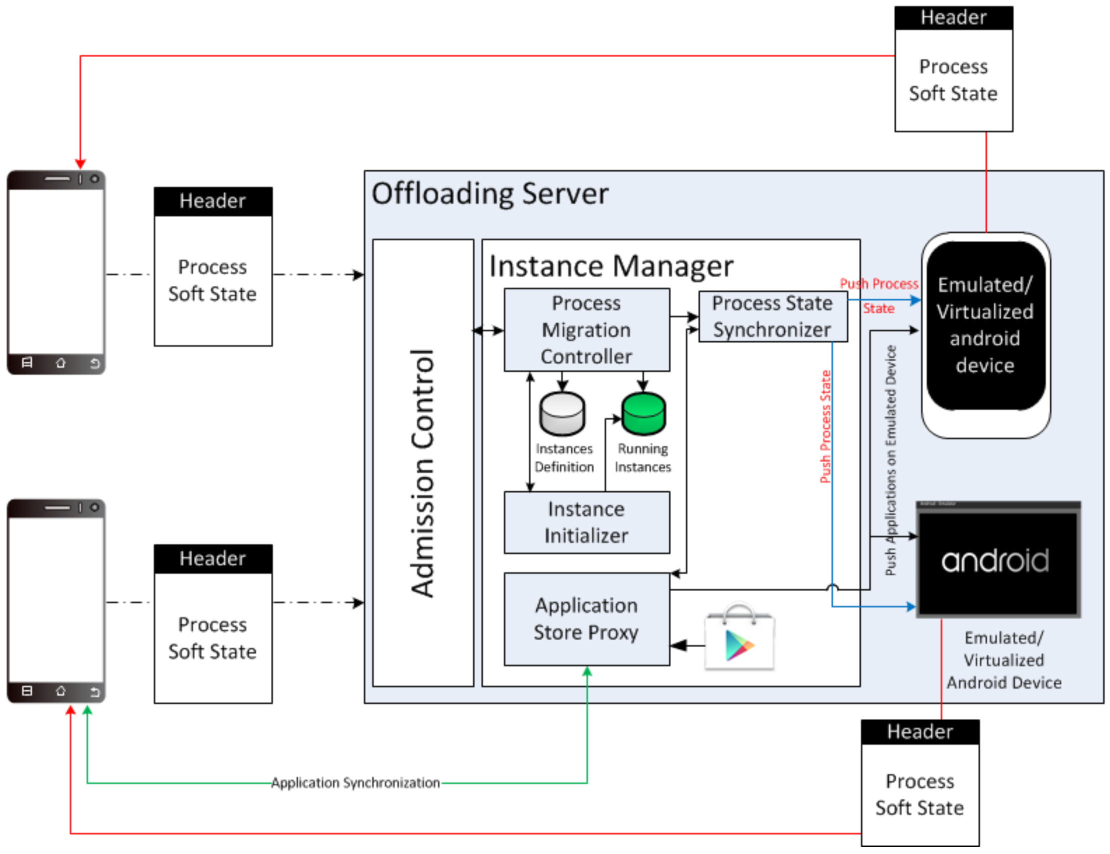


Figure 10: Proposed Mobile Cloud Computing Framework

The proposed framework exploits the concept of context switching in an operating system process and extends the basis of checkpoint/restart and process-migration mechanism. The process state of any computer program, as well as the ART-generated ELF file, is architecture dependent. A process state from an ARM-based machine cannot be used to restart the process on an x86-based machine. This condition is due to the difference in assembly, hardware components, instruction sizes, application binary interface, and other related factors. In most cases (almost 90%95%), mobile devices use an ARM-based machine [37, 38],], whereas the physical or virtual machines in the cloud are normally based on x86 architectures. Thus, we have three options for the process migration-based computational offloading for native and ART-based mobile applications. First, we can use a certain type of manual transformation of a process state from one architecture to another

[39]. We can disregard this option because it requires massive modification of the compiler or source code for both endpoints. Second, we can emulate the ARM instruction set in the cloud using binary translators, such as QEMU [40], which is not feasible for deadline and performance sensitive applications. Finally, we can utilize a compatible infrastructure (e.g., ARM) on the server. The availability of ARM infrastructure in the remote cloud or local cloudlet is crucial to actually envision the MCC ecosystem and enable phone clones and VMs in the remote servers.

We assume that the network conditions are stable, that no disconnection occurs during an offloading transaction, and that an offloading transaction is atomic. These assumptions are made to retain the focus on migration mechanism rather than network analysis and disconnection management, although these options should be explored in the future.

The description of the components on the mobile device and server as well as their interactions are described in the following sub-sections.

4.1. Mobile Devices Modules

On the mobile device, we have modified the original Android stack with an additional kernel module, which exploits the concept of context switching. Primarily, the module is configured to connect to the remote offloading server (i.e., VM in cloud) via an associated user space communicator, which also resides and is running on the mobile device. The connection parcel (Figure 11(a)) is composed of a device manifest and a software platform manifest.

The device manifest consists of the complete device profile, including the unique device ID, hardware information, and emulated/virtualized instance ID if the connection is established with the offloading server otherwise null. The software platform manifest will specify the operating system profile. If the connection to the remote offloading server is successful, then the kernel module residing in the device kernel will traverse all running applications and suspend any application that is being marked by the user as offloadable. After the suspension of applications, the module will serialize all the soft process states along with the packet header over the socket to the offloading server. The soft process state will include heap, thread local storage, stack, virtual memory areas, page backup, processor register contents, opened file states, and network socket states. The process state can be compressed with a lightweight compression mechanism to reduce the offloading parcel size. The header contains the device manifest and the application manifest (Figure 11(b)). The application manifest will contain the application profile, which is utilized by the instance manager to push a specific application from the central application repository to the virtualized/emulated phone running on the server. Finally, when a soft process state is received by the mobile device from the offloading server, the mobile device will extract the application manifest. The kernel module will then update the received process state for the extracted application manifest.

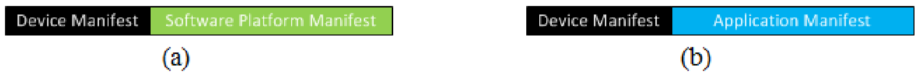


Figure 11: (a) Connection Parcel (b) Process state synchronization packet header

4.2. *Offloading Server Modules*

The offloading server is composed of loosely coupled components. The modules required at the server are described in the following sub-sections.

4.2.1. *Admission Control*

Admission control modules enforce the fair-usage policy of server resources. In case of a cloudlet with a weak pricing-model based admission control, ensuring the efficacy of the system is vital [41, 42]. Admission control can regulate the number of active server users depending on the system utilization or a manual policy defined by the system administrator. To share the resources among different devices, requests will be queued and processed using any feasible scheduling policy (e.g., FIFO, round robin) defined by the system administrator or dynamically selected based on the system load and other metrics. Furthermore, whenever a mobile device sends requests to the offloading server, the admission control will communicate with the instance manager to verify whether the device can be emulated/virtualized or not. Afterwards, the admission control verifies the availability of the software platform is available. If both conditions are satisfied, the request is transferred and the ID of the emulated/virtualized instance received from the instance manager is sent back to the requesting mobile device. Once the request is granted, the subsequent process state synchronization packets from that device are directly transmitted/forwarded to the instance manager.

4.2.2. *Instance Manager*

An instance manager manages the creation and deletion of emulated/virtualized Android instances. The instance manager is composed of four sub-components, namely, i) process-migration controller, ii) instance initializer, iii) application proxy store, and iv) process-state synchronizer.

Process-Migration Controller

The process-migration controller is the core component of the instance manager. This module communicates with the admission control upon a connection request from a mobile device. The module is also responsible for receiving process-state synchronization packets from the end-user mobile device. Upon receiving a message from the admission control, the controller queries the instance definition table to determine whether the requested device profile can be emulated/virtualized. If it can be emulated, then a query will be executed over the running instance store to find a running instance that can furnish the client request. If the query is successful, then a response with the existing running emulated/virtualized instance ID is sent to admission control, which acknowledges the end-user mobile device. If no running device is found, then the instance initializer is tasked to create an emulated/virtualized device whose ID is sent back to the user. Upon receiving a process-state synchronization parcel, the parcel is sent to the process-state synchronizer.

Instance Initializer

Upon receiving a device creation request from the process-migration controller, a new virtualized/emulated instance will be initialized with the specification from the device manifest and software platform manifest. The newly created emulated/virtualized device instance ID is acknowledged to the process-migration controller. An entry for this instance is then added in the running instance table.

Process-State Synchronizer

Upon receiving the process-state synchronization parcel from the controller, the synchronizer extracts the emulated instance ID and application manifest from the header. Then, it verifies whether an application with the extracted application manifest was pushed before or available at the extracted instance ID. If an application is available on the target device, the synchronizer pushes the checkpoint data received from the end-user mobile device to the emulated instance. If no application with the extracted application manifest is available at the target device, the synchronizer sends a message containing the extracted instance ID and the application manifest to the application store proxy. Subsequently, the synchronizer waits for a signal from the application store before executing the application on the device and then pushes the checkpoint data on the device. The module communicates with the emulated/virtualized device using an Android debug bridge and a communicator program installed and executed on the emulated/virtualized device upon initialization.

4.2.3. Application Store Proxy

The application store proxy acts as a cache for applications to be stored for subsequent requests. Upon receiving a request from the synchronizer, the application store proxy will check whether an application with that manifest exists in the cache. If available, the application store proxy will push the application to the emulated/virtualized device. Otherwise, the application can be downloaded using a method, such as that explained in [43]. If an application is unavailable in the local store and Google Play Store, another request is directly sent to the requesting device to share the application. Once an application is available in the local store, it is then pushed to the emulated instance, and the process-state synchronizer is acknowledged.

4.2.4. Module on Emulated/Virtualized Device

Similar with a mobile device, the emulated/virtualized Android device is configured with an additional kernel module and an associated user space communicator, which communicates with the end-user mobile device and other components in the offloading server (e.g., process-state synchronizer). When a soft process state is received by the emulated Android device from the end-user mobile device, it extracts the application manifest. The kernel module updates the received process state for the extracted application manifest. To send a process state to the end-user mobile device, the kernel module must use a method similar to that discussed in Section 4.1.

5. Proof of Concept

5.1. Empirical Setup

For a proof of concept, we deployed an ARM infrastructure box (Compulab Utilite[2]) at the server. We also deployed an x86 machine to host

[2]http://www.compulab.co.il/utilite-computer/web/utilite-overview

certain components of the proposed framework and foresee the ARM box. The topology of the experimental setup is presented in Figure 12

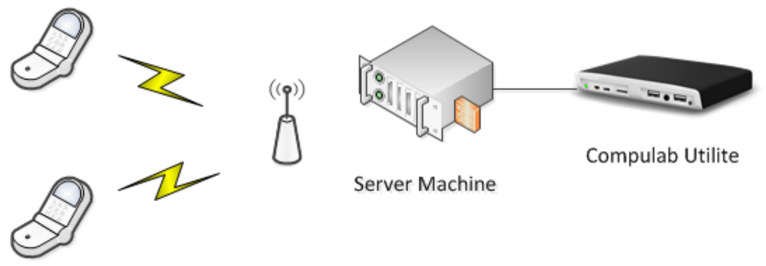


Figure 12: Experimental Setup Topology

The server PC configuration comprises Ubuntu 14.04, 4 GB Ram, and Core i7 CPU 3.4 GHz. A Samsung GT-I9100G smartphone is used as the client device. The server hosts the framework modules (i.e., application store proxy, instance manager, and admission control) to the ARM box. The proof-of-concept application is a 1000 x 1000 matrix multiplication program written in C with no migration or annotations and interfaces. The proof-of-concept application is also compiled with the standard Android toolchain downloaded along with the AOSP/CyanogenMod source code.

To capture the energy consumption of the application, we attached a hardware power meter to the battery of the smartphone (Figure 13). This step must be performed for two reasons. i) Smartphones do not offer a way to obtain fine-grained energy measurements of an application; (ii) The API of the smartphone does not capture power statistics for native applications.

5.2. Experiment Details

To migrate an application state from the client device (Samsung GT-I9100G) to the ARM box, both devices are flushed with the custom kernel to enable the insertion of kernel modules. Both devices (client device and ARM box) execute a user-space program to communicate with each other and transfer the process state over the socket. For the proof-of-concept experiment, the experimental applications are installed and executed on both devices. Once the communicators residing on the devices set up the link with each other, the modules on both sides detect the setup and start synchronizing the process state. However, the kernel modules periodically synchronize the applications executing on the other device because of the lack of synchronization markers in the experimental application. Thus, the migration

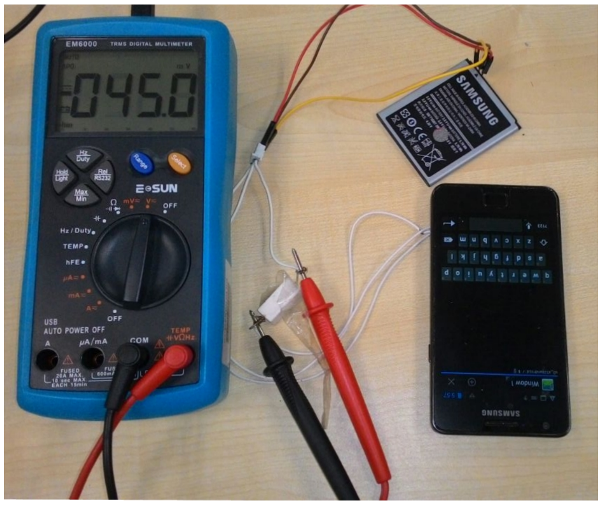


Figure 13: Hardware power meter setup used for energy measurements

control is unsuitable. We have performed 10 experiments and then averaged the results for presentations. Our first parameter of interest is the improvement in the execution time of the offloaded matrix multiplication application. Figure 14 presents the results of this parameter. The execution time of the offloaded application demonstrates a substantial improvement of approximately 76% compared with that of the baseline local execution. Similarly, the execution time of the offloaded application is reduced by almost 23% compared with that of the client server (SOA).

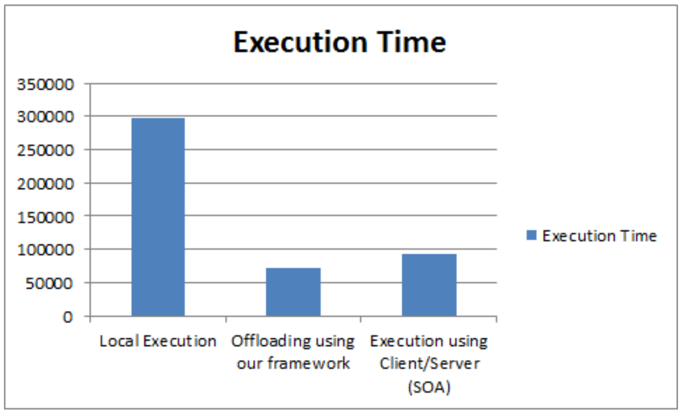


Figure 14: Execution Time Impact (in milliseconds)

The second important parameter is the energy consumption of the offloaded application. Similar to the analysis of the first parameter, this experiment is performed 10 times. The averages are presented in Figure 15. This experiment also shows the efficacy of the proposed framework compared with that of baseline local execution and SOA-based execution. However, the energy consumption of our proposed technique is more than that of the SOA-based execution because of the communication agent residing in the phone and extra bookkeeping, which ensures the consistency and accuracy of data offloading.

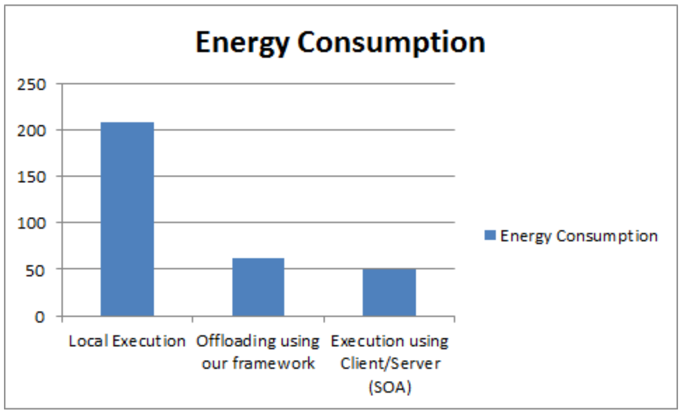


Figure 15: Energy Consumption Impact (in Joules)

The third important parameter is the number of bytes transferred between successive synchronizations of application states from the client device to the server ARM box. We make two cases for this parameter. First, we assume that client data is unavailable on the server and must be transferred to the server from the client device. Second, we assume that client data are available on the server. The results of the experiment are presented in Figure 16.

The number of bytes transferred in the first case (Figure 16) is approximately 12.2 MB per synchronization. The application declares and stores three 1000 1000 matrices (two are operands, one is the resultant). Thus, the memory acquired by the process for these three matrices are 12 MB (1,000,000 3 4 bytes). The bookkeeping of our framework adds an additional 200 KB to the memory. For the second case, only 206.4 KB is transferred for each synchronization interval. The data transfer amount in bytes can be further reduced by applying a differential-based mechanism used in VM migrations.

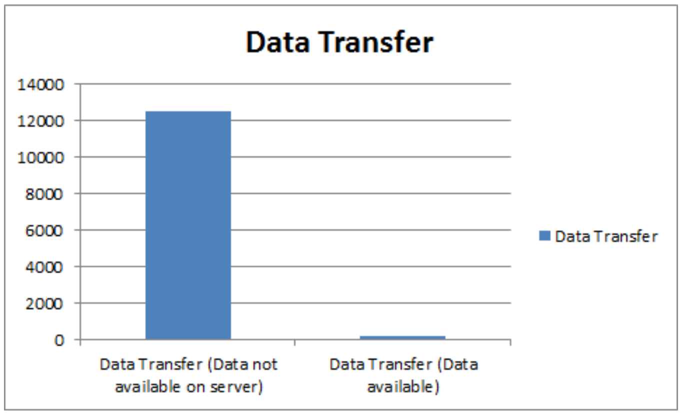


Figure 16: Data transfer (in bytes) between the client device and ARM box

5.3. Limitations of the proposed framework

The implementation of all the components of the devised framework is unfinished. Some of the components are at the initial development phase. Other components are implemented but needs improvement to handle the following issues to transform every mobile application to a mobile cloud application.

5.3.1. Improved Migration Mechanism

In the experimental setup of our framework, we capture and deploy the process state using Android kernel modules. The experiment reveals that using kernel module is not suitable for the migration for a number of reasons. They include code complexity, difference in kernel version between devices (in case of proximate mobile cloud), disabled kernel modules by default, need for a rooted and custom ROM on the phone to exploit this method, and the user curiosity on the security and privacy exploitation, which can be made in the kernel space. To handle these issues, an improved migration-based mechanism is required, which we will study in our future work.

5.3.2. Automatic Application Partitioning

All previous DVM-based partitioning mechanisms are incompatible with ART. To allow existing applications to take advantage of mobile cloud using computational offloading, these applications must be partitioned for the migration/synchronization points to be identified. These partitioning mechanisms, such as those used in CloneCloud [10] and its variants, should be automatic, to transform any mobile application to a mobile cloud application.

The partitioning mechanisms should work in such that when an application is installed on a mobile device using ART, the compilation process of ART should be modified to annotate and add special migration primitives to the code. The schematic of the original ART process and the envisioned modified ART process is presented in Figure 17 (a) and (b), respectively.

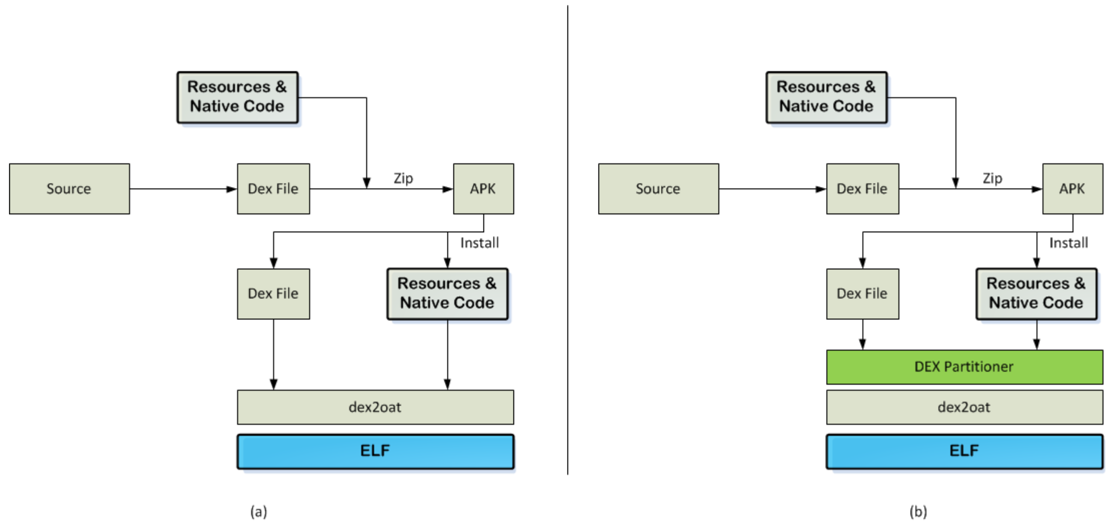
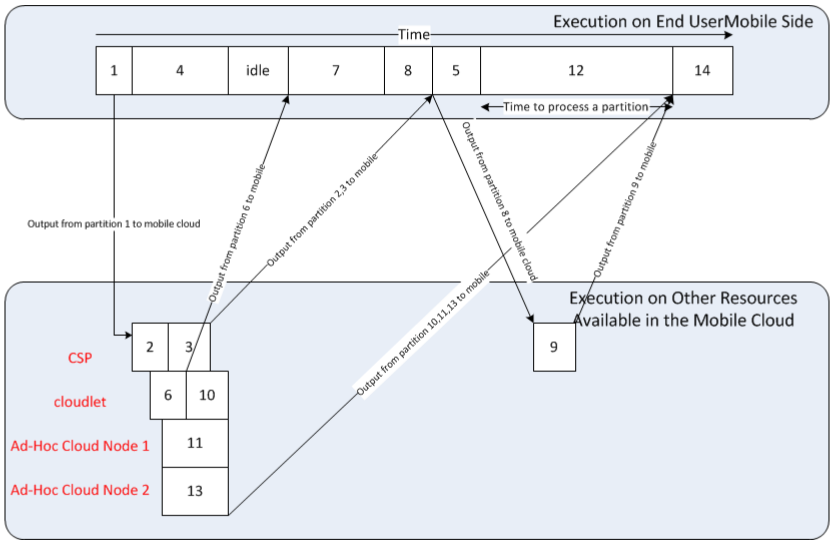


Figure 17: (a) The original Android Runtime Process (ART) (b) Modified ART Process

By inserting migration markers, the application can be migrated with great ease because the synchronization points are defined. The migration can be performed in a controlled manner by a user-level migration mechanism surpassing the kernel-level-state acquisition. The modified ART compilation process should be the same on both the mobile device and cloud to generate a similarly partitioned application.

5.3.3. Scheduling Partitions

In case the mobile cloud the user is connected is a complex mobile cloud scenario containing multiple type of resource as illustrated in Figure 1. The partitioning of the application in the above discussed manner (5.3.2) will help us to make a graph representation of the application which will be useful in formulating the scheduling problem. Once the partitioning of application is done, now two problems arise from the partitioned code i) to identify the dependencies between the partitions, (ii)schedule the offloading decisions/partitions over the resources available in the mobile cloud taking into account the partition dependencies (such that independent partitions can be executed in parallel) and other factors such as device context. A scheduling decision to whether offload a particular partition to CSP, cloudlet, or an ad-hoc mobile cloud node or not will be based on an objective function considering the dependencies between the partitions and the cost model. An illustration of such a schedule of an application with fourteen partitions on different types of resources available in the mobile cloud is presented in Figure 18.



Execution on End UserMobile Side
Time
1
4
idle
7
8
5
12
14
Time to process a partition
Output from partition 1 to mobile cloud
Output from partition 6 to mobile
Output from partition 2,3 to mobile
Output from partition 8 to mobile cloud
Output from partition 9 to mobile
Output from partition 10,11,13 to mobile
Execution on Other Resources Available in the Mobile Cloud
CSP
cloudlet
Ad-Hoc Cloud Node 1
Ad-Hoc Cloud Node 2
2
3
6
10
11
13
9

Figure 18: Schedule of automatically generated application partitions on mobile cloud resources

6. Conclusion

In this study, we have thoroughly examined the existing computational offloading primitives used in MCE. A computational offloading framework for native and ART-based mobile application is subsequently proposed. We conducted experiments to verify the efficacy of the proposed framework compared with that of baseline local execution and client-server models. The limitations of the study as well as future research directions are identified to transform every mobile application available on centralized application stores, such as the Google Play Store, into a mobile cloud application.

Acknowledgment

This work is fully funded and partially funded by Bright Spark Program and High Impact Research Grant from the University of Malaya under reference BSP/APP/1635/2013 and UM.C/625/1/HIR/MOE/FCSIT/03, respectively.

References

[1] J. Cohen. Embedded speech recognition applications in mobile phones: Status, trends, and challenges. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 5352–5355, March 2008.

[2] T. Soyata, R. Muraleedharan, C. Funai, Minseok Kwon, and W. Heinzelman. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000059–000066, July 2012.

[3] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Feng Xia, and Muhammad Shiraz. A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications*, 58:42–59, 2015.

[4] Abdullah Yousafzai, Victor Chang, Abdullah Gani, and Rafidah Md Noor. Multimedia augmented m-learning: Issues, trends and open challenges. *International Journal of Information Management*, 36(5):784 – 792, 2016.

[5] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[6] Abdullah Yousafzai, Abdullah Gani, Rafidah Md Noor, Mehdi Sookhak, Hamid Talebian, Muhammad Shiraz, and Muhammad Khurram Khan. Cloud resource allocation schemes: review, taxonomy, and opportunities. *Knowledge and Information Systems*, pages 1–35, 2016.

[7] Mahadev Satyanarayanan, P. Bahl, R Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, Oct 2009.

[8] Abdullah Yousafzai, Victor Chang, Abdullah Gani, and Rafidah Md Noor. Directory-based incentive management services for ad-hoc mobile clouds. May 2016.

[9] EGOR F. 64-bit android* and android run time. `https://software.intel.com/en-us/android/articles/64-bit-android-and-android-run-time`, 2014. (Visited on 06/30/2015).

[10] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[11] Yonggang Wen, Weiwen Zhang, and Haiyun Luo. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *INFOCOM, 2012 Proceedings IEEE*, pages 2716–2720, March 2012.

[12] Byung-Gon Chun and Petros Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, volume 9, pages 8–11, 2009.

[13] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[14] E.Y. Chen and M. Itoh. Virtual smartphone over ip. In *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a*, pages 1–6, June 2010.

[15] Bo Zhao, Zhi Xu, Caixia Chi, Sencun Zhu, and Guohong Cao. Mirroring smartphones for good: A feasibility study. In Patrick Snac, Max Ott, and Aruna Seneviratne, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 73 of *Lecture Notes of the*

Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 26–38. Springer Berlin Heidelberg, 2012.

[16] Shih-Hao Hung, Chi-Sheng Shih, Jeng-Peng Shieh, Chen-Pang Lee, and Yi-Hsiang Huang. Executing mobile applications on the cloud: Framework and issues. *Computers & Mathematics with Applications*, 63(2):573 – 587, 2012. Advances in context, cognitive, and secure computing.

[17] Jason Flinn and Z. Morley Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 84–89, New York, NY, USA, 2011. ACM.

[18] Ajay Surie, H. Andrés Lagar-Cavilla, Eyal de Lara, and M. Satyanarayanan. Low-bandwidth vm migration via opportunistic replay. In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications*, HotMobile '08, pages 74–79, New York, NY, USA, 2008. ACM.

[19] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA, 2013. IEEE Press.

[20] Ricky KK Ma and Cho-Li Wang. Lightweight application-level task migration for mobile cloud computing. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 550–557. IEEE, 2012.

[21] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953, March 2012.

[22] Soumya Simanta, Kiryong Ha, Grace Lewis, Ed Morris, and Mahadev Satyanarayanan. A reference architecture for mobile code offload in hostile environments. In David Uhler, Khanjan Mehta, and JenniferL. Wong, editors, *Mobile Computing, Applications, and Services*, volume 110 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 274–293. Springer Berlin Heidelberg, 2013.

[23] Dejan Kovachev, Yiwei Cao, and Ralf Klamma. Augmenting pervasive environments with an xmpp-based mobile cloud middleware. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 361–372. Springer Berlin Heidelberg, 2012.

[24] D. Kovachev, Tian Yu, and R. Klamma. Adaptive computation offloading from mobile devices into the cloud. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 784–791, July 2012.

[25] Byoung-Dai Lee. A framework for seamless execution of mobile applications in the cloud. In Zhihong Qian, Lei Cao, Weilian Su, Tingkai Wang, and Huamin Yang, editors, *Recent Advances in Computer Science and Information Engineering*, volume 126 of *Lecture Notes in Electrical Engineering*, pages 145–153. Springer Berlin Heidelberg, 2012.

[26] Tim Verbelen, Tim Stevens, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Dynamic deployment and quality adaptation for mobile augmented reality applications. *Journal of Systems and Software*, 84(11):1871 – 1882, 2011. Mobile Applications: Status and Trends.

[27] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Aiolos: Middleware for improving mobile application performance through cyber foraging. *Journal of Systems and Software*, 85(11):2629 – 2639, 2012.

[28] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: A computation offloading framework for smartphones. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer Berlin Heidelberg, 2012.

[29] H. Flores, S.N. Srirama, and R. Buyya. Computational offloading or data binding? bridging the cloud infrastructure to the proximity of the mobile user. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pages 10–18, April 2014.

[30] Salwa Adriana Saab, Farah Saab, Ayman Kayssi, Ali Chehab, and Imad H. Elhajj. Partial mobile application offloading to the cloud for energy-efficiency with security measures. *Sustainable Computing: Informatics and Systems*, pages –, 2015.

[31] Seungjun Yang, Donghyun Kwon, Hayoon Yi, Yeongpil Cho, Yongin Kwon, and Yunheung Paek. Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing. *Mobile Computing, IEEE Transactions on*, 13(11):2648–2660, Nov 2014.

[32] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *OSDI*, pages 93–106, 2012.

[33] Google. Art and dalvik. `https://source.android.com/devices/tech/dalvik/`, 2013. (Visited on 03/30/2015).

[34] Sean Buckley. Art experiment in android kitkat improves battery life and speeds up apps. `http://www.engadget.com/2013/11/06/new-android-runtime-could-improve-battery-life/`, 2013. (Visited on 03/30/2015).

[35] Tool Interface Standards Committee et al. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 2001.

[36] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. 2016.

[37] Forbes. Arm holdings and qualcomm: The winners in mobile. `http://www.forbes.com/sites/darcytravlos/2013/02/28/arm-holdings-and-qualcomm-the-winners-in-mobile/`, 2013. (Visited on 09/10/2015).

[38] Kristin Bent. Arm snags 95 percent of smartphone market, eyes new areas for growth. `http://www.crn.com/news/components-peripherals/240003811/arm-snags-95-percent-of-smartphone-market-eyes-new-areas-for-growth.htm`, 2012. (Visited on 09/10/2015).

[39] Kasidit Chanchio and Xian-He Sun. Data collection and restoration for heterogeneous process migration. *Software: Practice and Experience*, 32(9):845–871, 2002.

[40] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[41] M. Whaiduzzaman, A. Gani, and A. Naveed. Pefc: Performance enhancement framework for cloudlet in mobile cloud computing. In *Robotics and Manufacturing Automation (ROMA), 2014 IEEE International Symposium on*, pages 224–229, Dec 2014.

[42] Md Whaiduzzaman, Abdullah Gani, and Anjum Naveed. An empirical analysis of finite resource impact on cloudlet performance in mobile cloud computing. In *CEET-2014*. CEET, 2014.

[43] Dan Nanni. How to download apk files from google play store on linux. `http://xmodulo.com/download-apk-files-google-play-store.html`, 2015. (Visited on 08/30/2015).