Journal of KIIT. Vol. 15, No. 11, pp. 165-173, Nov. 30, 2017. pISSN 1598-8619, eISSN 2093-7571 165
http://dx.doi.org/10.14801/jkiit.2017.15.11.165

안드로이드 플랫폼을 사용한 스마트 디바이스를 위한 코드 오프로딩 프레임워크 설계 및 구현

이재훈*, 강수용**

Design and Implementation of Code Offloading Framework for Smart Devices Using Android Platform

Jae-Hun Lee*, Soo-Yong Kang**

본 연구는 미래창조과학부 및 정보통신기술진흥센터의 정보통신·방송 연구개발사업의 일환으로 수행하였음. [2014-0-00670, ICT 장비용 SW 플랫폼 구축]

요 약

스마트 폰, 태블릿 등 스마트 디바이스의 인기가 높아지고 수요가 늘어감에 따라 스마트 디바이스의 성능이 점차 좋아지고 있으며 이에 따라 모바일 어플리케이션이 제공하는 기능 또한 다양해지고 있다. 그러나 과거에 비해 월등히 향상된 성능에도 불구하고 스마트 디바이스는 여전히 부족한 컴퓨팅 자원과 배터리로 인해 어려움을 겪고 있다. 본 논문에서는 스마트 디바이스의 부족한 자원을 극복하기 위해 주변의 스마트 디바이스나 원격 서버로 컴퓨팅 집약적인 코드를 오프로딩해주는 Code Offloading Framework for Android(COFA)를 소개한다. 기존 연구들과는 달리 COFA는 클라우드 서버뿐만 아니라 주변 스마트 디바이스로 코드 오프로딩이 가능하며 오프로딩 정책보다는 방법에 중점을 두고 설계되었다. 또한 COFA가 목적대로 동작하는지 검증하기 위해 다양한 실험을 진행했으며 실험 결과를 통해 코드 오프로딩 기능을 효과적으로 제공하며 스마트 디바이스의 성능을 향상시켜줄 뿐만 아니라 에너지 소비를 감소시켜준다는 것을 확인할 수 있었다.

Abstract

As smart devices such as smart phones and tablets become more popular and demand grows, the performance of smart devices is gradually improving, and the functions provided by mobile applications are also diversifying. Despite significantly improved performance over the past, smart devices are still struggling with scarce computing resources and batteries. In this paper, we introduce Code Offloading Framework for Android (COFA), which offloads computing intensive code to nearby smart devices or remote servers to overcome scarce resources of smart devices. Unlike previous researches, COFA is not only able to offload the code to cloud server but also smart devices and is designed to focus on methods rather than offloading policy. In addition, we have conducted various experiments to verify that COFA works as intended, and we can confirm that the experimental results show that the code offloading function is effective, not only improves the performance of smart devices but also reduces energy consumption.

Keywords
mobile platform, smart device, code offloading, mobile cloud computing

* 한양대학교 컴퓨터소프트웨어학과
- ORCID: http://orcid.org/0000-0002-6511-4629
** 한양대학교 컴퓨터소프트웨어학과 교수 (교신저자)
- ORCID: http://orcid.org/0000-0002-5332-7846

· Received: Oct. 09, 2017, Revised: Oct. 30, 2017, Accepted: Nov. 02, 2017
· Corresponding Author: Soo-Yong Kang
Dept. of Computer Science, Hanyang University
222, Wangsimni-ro, Seongdong-gu, Seoul, Korea.
Tel.: +82-2-2220-1725, Email: sykang@hanyang.ac.kr

Ⅰ. 서 론

스마트 폰, 태블릿 등의 스마트 디바이스가 보편화되면서 모바일 어플리케이션 또한 다양해지고 복잡해졌다. 이와 같은 어플리케이션들을 원활하게 사용할 수 있도록 고 사양 컴퓨팅 파워를 지닌 스마트 디바이스가 등장하고 있다. 그러나 스마트 디바이스의 처리 능력이 향상되더라도 서버의 처리 능력에 비해서는 아직 부족하다[1]. 또한 스마트 디바이스의 컴퓨팅 파워는 배터리 수명, 디바이스 크기, 발열 감소 등의 이유로 우선순위에서 밀리기도 한다[2]. 심지어 구글 안경, Pebble 시계와 같은 소형 스마트 디바이스는 스마트 폰보다 매우 낮은 컴퓨팅 능력을 갖고 있다[3].

모바일 디바이스의 제한된 컴퓨팅 자원을 극복하기 위해 외부에 존재하는 서버를 사용하고자 하는 다양한 연구가 진행되었다. 스마트 디바이스의 작업을 클라우드 서버에 전송해 실행하는 모바일 클라우드[4]-[6]가 대표적인 접근법이라 할 수 있다. 그러나 이 접근법은 어플리케이션 코드 일부를 서버에서 실행하기 위해 개발자가 신경 써야 할 부분이 많다는 단점이 있다. 서버에서 실행될 부분을 정해야 하고 해당 코드가 서버에서 동작될 수 있도록 서버에서 제공하는 틀에 맞춰서 작성해야 한다. 또한 네트워크 연결 상태가 좋지 않거나 사용할 수 없는 경우 서버의 도움을 받는 것이 오히려 성능 저하를 발생시킬 수 있다. 이런 문제점을 해결하기 위해 어플리케이션 코드 일부를 자동적으로 원격 서버에 전송 및 실행해주는 코드 오프로딩 접근 방식이 등장했다.

코드 오프로딩 접근 방식은 많은 컴퓨팅 자원을 요구하는 코드를 원격 서버에 실행시킴으로써 어플리케이션 성능과 스마트 디바이스 배터리 수명을 향상시키게 도와준다. 그러나 이런 장점에도 불구하고 모바일 플랫폼 수준에서 코드 오프로딩을 제공하기 위한 시스템은 부족하다. 이로 인해 어플리케이션이 코드 오프로딩 기능을 사용하기 위해서는 클라이언트/서버 역할을 하는 어플리케이션 또는 서비스가 필요할 뿐만 아니라 기존 어플리케이션의 수정이 불가피하다. 만약 플랫폼 수준에서 코드 오프로딩 기능을 제공한다면 기능 제공을 위한 추가적인 어플리케이션/서비스의 구현이 필요 없을 뿐만 아니라 기존 일반 어플리케이션도 약간의 코드 수정만으로 코드 오프로딩 기능을 사용할 수 있을 것이다.

본 논문에서는 COFA(Code Offloading Framework for Android)를 소개한다. COFA는 어플리케이션의 컴퓨팅 집약적인 코드를 분리하고 원격 서버 또는 주변 스마트 디바이스에 적재 및 실행시키는 기능을 플랫폼 수준에서 제공한다. 또한 COFA는 코드 오프로딩을 사용하기 위해 최소한의 코드 수정만을 요구하며 코드 캐쉬(Cache)를 통해 네트워크 부하를 최소화했다. 이 시스템은 코드 포장, 선택적 사전 전송 기법을 통해 코드 오프로딩 기능을 제공한다.

코드 포장 기법은 어플리케이션 코드 내 컴퓨팅 집약적인 부분을 추출해 원격 디바이스에서 실행 가능한 형태로 포장하는 과정이다. 어플리케이션 설치 시 패키지 내 실행파일을 코드 단위로 분해 후 개발자가 표시한 부분을 모아 실행파일로 치환한다. 선택적 사전 전송 기법은 어플리케이션 실행 시 원격 디바이스에서 실행될 코드를 판단하고 해당 코드가 실행될 수 있도록 필요한 데이터를 미리 전송하는 기법이다. 전송된 코드는 코드 캐쉬에 저장돼 같은 코드 수행 시 추가적인 코드 전송 없이 실행이 가능하며 이로 인해 네트워크 부하를 최소화할 수 있다.

COFA가 의도대로 동작하는지 검증하기 위해 안드로이드 플랫폼 내 프로토타입을 구현 후 다양한 실험을 진행했다. 실험으로 COFA를 사용해 주변 스마트 디바이스 또는 서버로 코드 오프로딩 시 큰 폭으로 성능이 향상되며 소비되는 에너지를 절약할 수 있다는 것을 확인할 수 있었으며 이를 통해 COFA가 코드 오프로딩 기능을 효과적으로 제공한다는 것을 검증할 수 있었다.

본 논문은 COFA의 디자인과 구현을 담고 있다. 2장에서는 스마트 디바이스의 제한된 자원을 극복하기 위해 진행된 다양한 연구에 대해 소개한다. 3장에서는 COFA의 구조 및 실행 흐름과 함께 코드 포장, 선택적 사전 전송 기법에 대해 설명한다. 4장에서는 COFA의 성능을 검증하기 위한 실험의 결과를 서술하고 마지막으로 5장에서 결론을 맺는다.

II. 관련 연구

스마트 디바이스의 제한된 자원을 극복하기 위한 연구는 다양한 분야에서 진행되고 있다. 클라우드 서버의 도움을 받아 작업을 처리하는 모바일 클라우드 컴퓨팅이 대표적인 관련 분야의 대표적인 예라 할 수 있다. 본 논문에서는 모바일 클라우드 컴퓨팅 연구 중 어플리케이션 전체 또는 일부를 클라우드 서버에 오프로딩 함으로써 성능 향상을 야기하고자 하는 연구를 소개한다.

MAUI[7]는 최소한의 어플리케이션 수정만으로 작은 단위의 코드 오프로딩을 제공하며 어플리케이션 상태 전송을 최소화하는 것이 성능 향상에 큰 영향을 준다는 것을 보여주었다. 그러나 코드 실행에 필요한 정보를 XML으로 전송해 직렬화 부하가 크다는 단점이 있다. CloneCloud[8]는 어플리케이션을 실행중인 가상 머신(Virtual Machine)을 클라우드 서버에 오프로드 하는 접근 방식을 제안했다. 운영체제 수준에서 오프로딩을 제공하므로 어플리케이션의 수정 없이 오프로딩 기능을 사용할 수 있다. 그러나 가상 머신을 통째로 전송하므로 네트워크 부하가 클 뿐만 아니라 클라우드 서버에 모바일 디바이스와 같은 설정을 가진 하드웨어 시뮬레이터가 실행 중이어야만 동작한다. Zhang et al의 연구[9]에서는 스마트 디바이스의 에너지 소비를 줄이기 위한 오프로딩 정책을 제시했으며, Kumar[10]는 정적 네트워크 환경에서 컴퓨팅 에너지 분석 모델을 통해 오프로딩으로 절약할 수 있는 에너지를 계산하는 측정하는 방법을 제시했다.

위 연구들 외에도 오프로딩 접근 방식을 통해서 스마트 디바이스의 제한된 자원을 극복하고자 하는 연구[11]-[13]는 계속 진행 중이다. 그러나 이러한 연구들의 대부분은 오프로딩 여부를 판단하는 정책에 중점을 두고 있으며, 클라우드 서버로만 오프로딩이 가능하다. 본 논문에서 제안하는 COFA는 오프로딩 정책 보다는 오프로딩 방법에 중점을 두고 있으며 서버뿐만 아니라 주변의 스마트 디바이스로도 오프로딩이 가능하다.

III. COFA: Code Offloading Framework for Android Platform

COFA의 핵심 기능은 어플리케이션의 코드 일부를 주변 디바이스 또는 서버에서 실행할 수 있도록 하는 것이다. 본 논문에서는 모든 안드로이드 디바이스에 COFA 기술이 탑재되어 있고 어플리케이션은 한 디바이스에만 설치된 상태라 가정한다. 이 때 어플리케이션이 설치되어 있는 디바이스를 로컬 디바이스, 코드 일부를 받아 수행할 디바이스/서버를 원격 디바이스/서버라 한다.

3.1 안드로이드 배경

안드로이드 플랫폼은 어플리케이션 설치를 위해 Android Application Package(APK) 파일을 사용한다. 사용자로부터 어플리케이션 설치 명령이 내려지면 어플리케이션 설치를 담당하는 시스템 서비스인 PackageManagerService는 APK 파일을 분해해 AndroidManifest.xml로 어플리케이션 정보를 등록하고 실행파일(classes.dex)과 리소스(이미지, 레이아웃 등)를 추출해 적절한 위치에 배치함으로써 어플리케이션을 설치한다.

그림 1은 APK 파일이 생성되는 과정을 나타낸다. java 소스 파일들은 class 파일들로 컴파일 된 후 안드로이드 라이브러리의 class 파일들과 함께 실행파일인 dex 파일로 변환된다. 또한 이미지, 레이아웃 등의 리소스는 aapt(Android Asset Packaging Tool)를 통해 컴파일 된 후 테이블 형태로 치환되어 resources.arsc라는 파일로 저장된다. 이렇게 생겨난 dex 파일과 resources.arsc 파일을 함께 압축해 APK 파일이 생성된다.

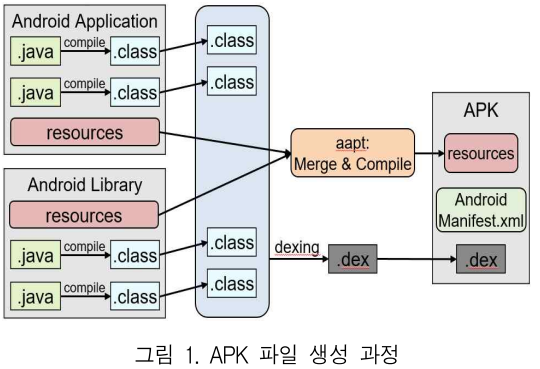

그림 1. APK 파일 생성 과정
Fig. 1. APK file creation process

3.2 구조 및 실행 흐름

그림 2는 COFA를 위해 안드로이드 플랫폼 또는 서버에 추가된 구성요소들을 나타낸다. 안드로이드 플랫폼을 사용하는 디바이스만 로컬 디바이스 또는 원격 디바이스가 될 수 있으며 원격 서버의 경우 JVM(Java VM)만 실행 가능하면 운영체제에 상관없이 COFA 서버 역할을 수행할 수 있다. 별표(*)로 표시된 부분은 기존 구성요소 중 수정된 부분을 나타내며 회색으로 표시된 부분은 추가된 구성요소를 의미한다. COFA를 사용한 일반적인 코드 오프로딩 과정은 다음과 같다.

어플리케이션이 설치되는 과정에서 해당 어플리케이션이 코드 오프로딩 기능을 사용하는 경우 PackagingService는 APK 파일을 분해 후 코드 포장 기법을 통해 컴퓨팅 집약적인 코드를 포장한다(①). 그 후 어플리케이션이 실행되면 선택적 사전 전송 기법을 통해 포장된 코드를 전송해 적절한 위치에 저장한다(②). 실행된 어플리케이션이 컴퓨팅 집약적인 코드를 호출하면 OffloadingService는 이를 가로챈다(③). 그 후 DecisionEngine은 연결된 원격 디바이스/서버의 상태를 파악해 오프로딩 여부를 결정한다. 오프로딩이 결정되면 대상으로 결정된 원격 디바이스/서버로 호출된 메소드의 이름 및 인자와 패키지 이름을 전송한다(④). 원격 디바이스/서버의 TaskExecutor는 전송받은 정보를 사용해 코드를 실행 후 결과를 돌려준다(⑤).

3.3 코드 포장 기법

어플리케이션 코드 일부를 다른 디바이스에서 실행시키기 위해서 떠올릴 수 있는 가장 쉬운 방법은 어플리케이션 코드 전체를 전송해 실행하는 것이다. 그러나 이 경우 데이터 전체가 전송되므로 전송 시간이 오래 걸릴 뿐만 아니라 서버 디바이스에서 실행될 수 있도록 설치하는 과정이 필요하다. COFA에서는 코드 포장 기법을 통해 어플리케이션의 컴퓨팅 집약적 코드를 추출해 개별적으로 관리함으로써 이를 방지한다.

코드 포장 기법은 안드로이드 어플리케이션의 실행파일을 분해해 분석 과정을 통해 컴퓨팅 집약적인 코드를 추출 후 원격 디바이스/서버에서 실행할 수 있는 형태로 포장하는 과정이며 그림 3과 같이 동작한다. 이를 위해서는 우선 어플리케이션의 실행파일인 classes.dex 파일을 코드 수준으로 분해해야 한다. dex 파일은 역 컴파일(Decompile) 과정을 통해 java class 파일들로 변환이 가능하지만 class 파일은 사람이 해석할 수 없는 형식으로 되어 있어 분석이 불가능하다.

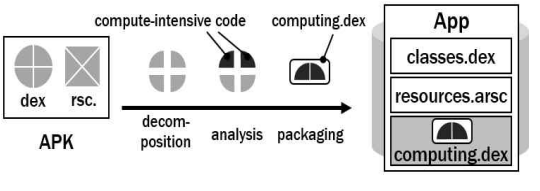

그림 3. 코드 포장 기법 진행 과정
Fig. 3. Code packaging process

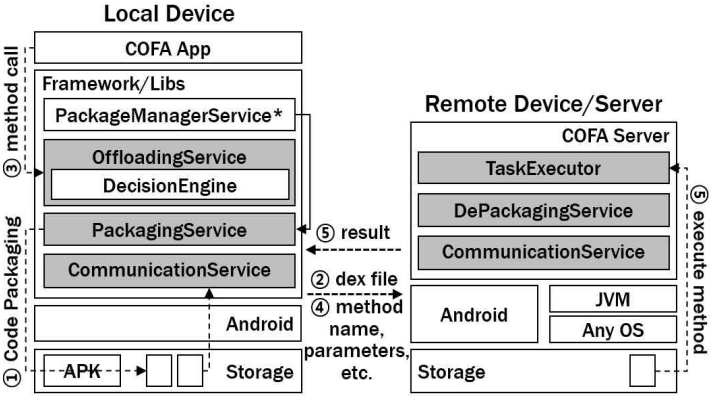

그림 2. 안드로이드 플랫폼 내 COFA 구성요소
Fig. 2. COFA components in android platform

사람이 해석할 수 있는 형식으로 되어 있는 smali 파일을 사용해 이 문제를 해결했으며 dex 파일로부터 smali 파일을 추출하기 위해 baskmali tool [14]을 안드로이드 플랫폼용으로 포팅해 사용했다.

이렇게 분해된 smali 파일 중 오프로딩의 대상이 되는 코드를 분류하기 위해 [7], [11]과 비슷한 자바 어노테이션 접근 방식을 사용한다. 개발자가 원격 디바이스/서버로 오프로딩 하고자 하는 코드를 보유한 클래스에 *@ComputeIntensive*라는 어노테이션을 명시하면 해당 클래스의 smali 파일은 오프로딩 대상 코드로 분류된다. 어플리케이션 실행 중 해당 클래스가 보유한 메소드가 호출되면 미리 정의된 AOP(Aspect-Oriented Programming) 프레임워크[15]는 메소드 호출을 가로채 OffloadingService로 전달한다. @ComputeIntensive 어노테이션이 명시된 코드를 파악하기 위해 dex 파일로부터 얻은 smali 파일들을 파싱하며 이 과정에서 오프로딩 대상이 되는 코드가 참조하는 패키지를 파악해 코드 테이블을 작성한다. 코드 테이블은 2열로 구성된 표로 오프로딩 대상 코드별로 해당 코드가 사용하는 패키지 목록을 나타낸다.

smali 파일들의 분석을 통해 컴퓨팅 집약적 코드와 해당 코드가 사용하는 패키지들이 분류되면 원격 디바이스/서버에서 실행 가능한 형태로 포장한다. 이는 smali 파일들을 모아 dex 파일로 변환하는 과정인데 이 과정에서 안드로이드 플랫폼용으로 포팅된 smali tool[14]을 사용한다.

코드 포장 기법은 실행 파일 분해, 분석 등의 작업으로 인해 오버헤드가 상당하다. 또한 오프로딩 기능을 사용하지 않는 어플리케이션에 대해 코드 포장을 진행할 경우 성능 저하를 발생시킬 수 있다. 이를 방지하기 위해 안드로이드 어플리케이션 설정 파일인 AndroidManifest.xml 내부의 <Application> 요소에 사용할 수 있는 *useOffloding* 속성을 추가했다. 어플리케이션 설치 시 PackageManagerService는 <Application>요소 내 “android:useOffloading=true” 항목을 발견하면 오프로딩 기능을 사용하는 어플리케이션이라 판단하고 PackagingService로 어플리케이션의 정보를 전달한다. 그 후 PackagingService는 해당 어플리케이션에 대해 코드 포장을 진행한다. 또한 코드 포장 과정은 어플리케이션 설치 시 한번만 진행되므로 어플리케이션이 실행된 후 코드를 주고받는 동안에는 성능 저하를 발생시키지 않는다.

3.4 선택적 사전 전송 기법

어플리케이션이 오프로딩 대상 코드를 호출했을 때 해당 코드를 전송하면 직관적이지만 네트워크 통신 오버헤드로 인해 지연이 발생한다. 이를 방지하기 위해 COFA는 오프로딩 기능을 사용하는 어플리케이션이 실행되면 연결된 원격 디바이스/서버에 해당 어플리케이션의 오프로딩 해당 코드(dex 파일)를 미리 전송한다. 이렇게 전송된 코드는 로컬 디바이스와 연결이 지속되는 동안 램디스크(ramdisk)로 구성된 코드 캐쉬에 유지되며 연결이 끊어질 때 삭제된다. 만약 이미 전송되어 있다면 다시 전송하지 않고 기존에 전송된 파일을 사용한다. 이 과정을 선택적 사전 전송이라 하며 이를 통해 메소드 호출 때 마다 코드를 전송할 필요 없이 코드 오프로딩이 가능하다.

3.5 오프로딩 결정

COFA는 오프로딩 여부와 연결된 디바이스/서버가 여러 대일 경우 어떤 디바이스로 오프로딩 할 것인지 결정하기 위해 간단한 결정 엔진을 사용한다. 원격 디바이스 또는 서버가 연결되면 로컬 디바이스의 DecisionEngine은 미리 정의된 테스트 코드를 전송해 실행시킨 후 시간을 기록한다. 이 시간이 로컬 디바이스의 테스트 코드 수행 시간보다 작다면 해당 디바이스로 오프로딩 한다. 만약 연결된 디바이스의 수가 여러 대라면 실행 시간이 가장 짧은 디바이스로 오프로딩 명령을 내린다. 향후에는 더 나은 오프로딩 결정을 위해 개발자가 COFA의 결정 엔진을 확장시킬 수 있도록 할 계획이다.

3.6 원격 디바이스/서버에서의 코드 실행

로컬 디바이스로부터 오프로딩 될 코드는 dex 파일로 되어 있다. 안드로이드 플랫폼에서는 어플리케이션의 크기가 커지는 것을 방지하기 위해 실행 파일을 여러 dex 파일로 나눠 실행할 수 있는 기능을

제공한다. 이는 하나의 메인 실행 파일을 두고 나머지를 라이브러리처럼 사용하도록 하는 방법으로 DexClassLoader를 통해 이뤄진다. 원격 디바이스의 경우 로컬 디바이스와 같이 안드로이드 플랫폼을 사용하므로 전송받은 코드를 DexClassLoader를 사용해 실행 후 결과를 돌려준다.

원격 서버의 경우 안드로이드 플랫폼을 사용하지 않아 dex 파일을 실행하기 위해서는 추가적인 작업이 필요하다. 안드로이드 플랫폼을 적재한 가상 머신을 사용하면 원격 서버를 안드로이드 디바이스처럼 사용할 수 있지만 가상 머신을 통해 실행시키는 것이므로 성능 저하가 발생한다. 이를 방지하기 위해 DePackagingService는 dex 파일을 전송받았을 때 dex2jar[16]를 사용해 jar 파일로 변환 후 저장한다. 그 후 컴퓨팅 집약적인 코드가 호출되면 원격 디바이스에서와 마찬가지로 ClassLoader를 사용해 코드를 실행한다.

Ⅳ. 성능 평가

4.1 실험 환경

COFA의 성능을 평가하기 위해 로컬 디바이스로는 LG에서 생산한 Nexus 7, 원격 디바이스로는 Nexus 6P, 원격 서버로는 PC에 리눅스 서버를 탑재해 사용했다. AOSP 6.0.1_r11[17] 내부에 COFA 프로토타입을 구현해 Nexus 7, Nexus 6P에 탑재했으며 실험에 사용한 디바이스/서버의 스펙은 표 1과 같다. COFA를 통해 코드 오프로딩을 함으로써 성능 향상과 에너지 절약할 수 있다는 사실을 검증하기 위해 두 가지 어플리케이션을 구현 후 실험을 진행했다.

표 1. 안드로이드 디바이스/리눅스 서버 스펙
Table 1. Android device and linux server specifications

	Nexus 7	Nexus 6P	Server PC
CPU	Quad-core 1.2GHz Cortex-A9	4x1.55GHz Cortex-A53 & 4x2.0GHz Cortex-A57	Intel(R) Core(TM) i7-6700 3.40GHz
Memory	1GB	3GB	16GB
OS	AOSP 6.0.1	AOSP 6.0.1	Ubuntu 16.04

실험에 사용된 디바이스/서버는 Wifi를 사용해 연결되어 있으며 Wifi AP는 802.11 a/b/g/n/ac를 지원한다. 로컬 디바이스인 Nexus 7의 에너지 소비를 측정하기 위해 */sys/class/power_supply/battery* 폴더 내 *current_now* 파일과 *voltage_now* 파일의 값을 주기적으로 측정해 사용했다.

4.2 오델로(Reversi)

실험에 사용한 첫 번째 어플리케이션은 오델로(Reversi)로 minimax 알고리즘[18]을 사용해 다음 수를 계산한다. 계산의 복잡성을 위해 minimax tree의 깊이 6까지 탐색한 후 수를 결정하도록 구현했다. 또한 수를 두는 시간까지의 지연을 방지하기 위해 두 플레이어 모두 인공지능을 사용해 자동적으로 수를 계산하도록 했다. 다음 수를 계산하는 코드를 오프로딩 대상 코드로 설정해 실험을 진행했고 5회 반복 실험 후 평균값을 측정했다.

그림 4는 로컬 디바이스(Nexus 7) 홀로 오델로 게임을 실행했을 때와 원격 디바이스(Nexus 6P) 또는 원격 서버로 코드 오프로딩 했을 때의 실행 시간을 나타낸다. UI는 수를 계산 후 유저 인터페이스에 결과를 반영하는 시간을 나타내며 Computation은 수를 계산하는 시간을 나타낸다. 원격 디바이스/서버의 도움을 받을 때 다음 수를 계산하는 Computation 시간에서는 성능이 향상되는 반면 UI 반영 시간은 일정하게 나타나는 것을 알 수 있다. 전체적인 성능을 보았을 때 원격 디바이스로 코드 오프로딩 시 1.8배, 원격 서버로 코드 오프로딩 시 4.7배 성능이 향상했으며 Computation 부분만 측정했을 시 각각 2.2, 13.3배 성능이 향상되었다는 것을 알 수 있다.

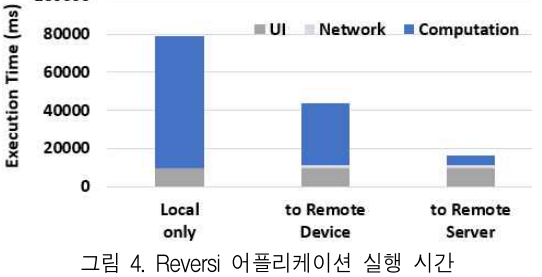

그림 4. Reversi 어플리케이션 실행 시간
Fig. 4. Execution time for Reversi application

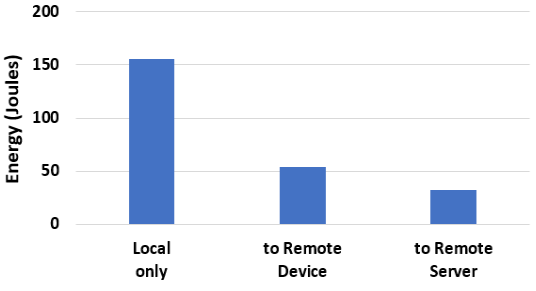

그림 5. Reversi 어플리케이션 에너지 소비량
Fig. 5. Energy consumption for Reversi application

그림 5는 오델로 어플리케이션 실행 시 소비되는 에너지를 나타낸다. 실험 결과를 통해 COFA를 사용해 컴퓨팅 집약적인 코드를 오프로딩 시 에너지가 절약되는 것을 확인할 수 있다. 원격 디바이스로 오프로딩 시에는 65%, 원격 서버로 오프로딩 시에는 79% 에너지 소비가 감소되는 것을 확인할 수 있으며 원격 서버로 전송할 때가 원격 디바이스로 전송할 때 보다 전체 실행 시간이 짧아 더 적은 에너지를 소비하는 것을 확인할 수 있다.

4.3 린팩 벤치마크(Linpack Benchmark)

두 번째로는 디바이스 성능을 정량적으로 나타내기 위해 많이 사용되는 린팩 벤치마크를 안드로이드 어플리케이션으로 구현해 실험을 진행했다. 린팩 벤치마크는 n x n 행렬의 선형대수 처리를 통해 시스템의 부동소수점 계산 능력을 평가한다[19]. 부동소수점을 계산하는 코드를 오프로딩 대상으로 하여 실험을 진행했으며 오델로 어플리케이션 실험과 마찬가지로 5회 실험 후 평균값을 측정했다.

배열의 크기(n)를 다양하게 조절해가며 린팩 벤치마크 성능을 측정했으며 그 결과는 그림 6과 같다. n의 사이즈가 커질수록 로컬 디바이스 홀로 수행했을 때와 코드를 오프로딩 했을 때의 차이가 커지는 것을 확인할 수 있다. 벤치마크 결과 평균적인 처리량은 로컬 디바이스 홀로 처리 시 106.1 MFLOPS, 원격 디바이스로 코드 오프로딩 시 298.5 MFLOPS, 원격 서버로 오프로딩 시 2719.6 MFLOPS였으며 원격 디바이스/서버의 도움으로 인해 각각 2.9, 29.2배 성능이 향상되었다.

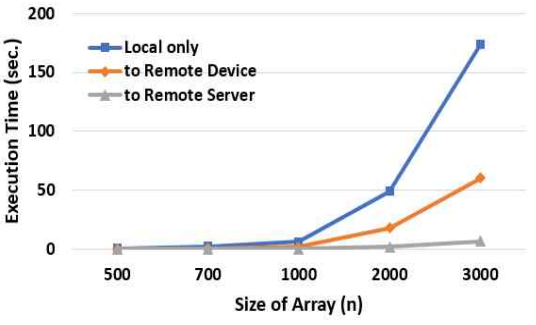

그림 6. 린팩 어플리케이션 실행 시간
Fig. 6. Absolute execution time for Linpack application

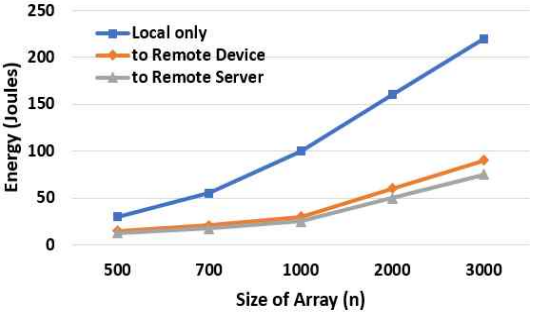

그림 7. 린팩 어플리케이션 에너지 소비량
Fig. 7. Energy consumption for Linpack application

그림 7은 린팩 어플리케이션을 통한 벤치마크 실행 도중 소비되는 에너지를 측정한 그래프이다. 배열의 크기가 커질수록 계산하는 양이 늘어 소비되는 에너지가 증가하는 것을 확인할 수 있으며 배열의 크기가 3000일 때 로컬 디바이스 홀로 수행했을 때 보다 원격 디바이스로 오프로딩 시 59%, 원격 서버로 오프로딩 시 66% 감소된 에너지가 소비되는 것을 확인할 수 있었다.

4.4 어플리케이션 설치 시간

코드 오프로딩 기능을 사용하는 어플리케이션은 설치 시 코드 포장을 위해 추가적인 과정(분해, 분석, 포장)을 거친다. 코드 포장 과정의 부하를 측정하기 위해 어플리케이션 설치 시간을 측정했다. 그림 8은 오델로와 린팩 어플리케이션 설치 시 소요되는 측정한 그래프이다. 설치 외 세 가지 항목은 코드 포장 기법으로 인해 소요되는 시간이다.

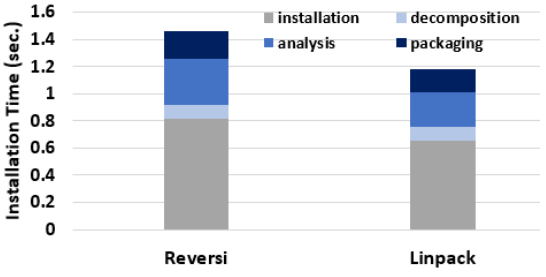

그림 8. 어플리케이션 설치 시간 세분화
Fig. 8. Application installation time breakdown

*분해*는 baksmali tool을 사용해 dex 파일을 smali 파일들로 분해하는데 소요된 시간을 나타내고 *분석*은 smali 파일들을 분해해 컴퓨팅 집약적인 코드를 추리는 시간, *포장*은 추려진 코드들을 모아 smali tool을 사용해 다시 dex 파일로 변환하는 시간을 의미한다. 두 어플리케이션 모두에서 코드 포장을 수행함으로써 설치만 하는 시간 대비 80%가 증가한 시간이 소요되었다. 그러나 코드 포장으로 인해 소요되는 시간을 포함하더라도 전체 설치 시간은 1.5초미만으로 길지 않은 시간이다. 또한 코드 포장 과정은 어플리케이션 설치 시 한번만 진행되고 그 이후 코드 오프로딩 과정에서는 실행되지 않는다. 그러므로 코드 포장 기법으로 인한 부하는 코드 오프로딩 성능에는 영향을 주지 않는다 할 수 있다.

V. 결론 및 향후 과제

본 논문에서는 스마트 디바이스의 제한된 자원을 극복하기 위해 주변 디바이스 또는 원격 서버에 컴퓨팅 집약적 코드를 오프로딩 해주는 COFA를 소개했다. 또한 COFA가 의도대로 동작하는지 검증하기 위해 안드로이드 플랫폼 내 프로토타입을 구현하고 다양한 실험을 진행하였다.

평가를 통해 스마트 디바이스로 코드를 수행할 때 보다 COFA를 통해 원격 디바이스/서버로 코드 오프로딩 시 성능이 향상될 뿐만 아니라 에너지를 절약할 수 있다는 사실을 검증할 수 있었다. 이로 인해 코드 오프로딩은 효과적으로 스마트 디바이스의 제한된 자원을 극복할 수 있는 방법이며 COFA는 그 목적에 맞게 동작한다는 것을 증명했다.

더욱 효과적이고 효율적인 코드 오프로딩을 제공하기 위해 개발자가 COFA 오프로딩 결정 엔진에 정책을 추가할 수 있도록 개선할 예정이다. 또한 느린 네트워크 및 디바이스의 낮은 성능으로 원격 디바이스/서버와의 연결이 불안정한 스마트 디바이스의 특성을 극복하기 위해 실시간 생존 검사와 작업 이전 기법 등의 결함 허용 기법을 추가할 예정이다.

References

[1] Jason Flinn, "Cyber foraging: Bridging mobile and cloud computing", Synthesis Lectures on Mobile and Pervasive Computing, Vol. 7, No. 2, pp. 1-103, Sep. 2012.

[2] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan, "Just-in-Time provisioning for cyber foraging", CMU School of Computer Science, Tech. Rep. CMU-CS-12-148, pp. 153-166, Dec. 2012.

[3] Afnan Fahim, Abderrahmen Mtibaa, and Khaled A Harras, "Making the case for computational offloading in mobile device clouds", MobiCom '13: Proceedings of the 19th annual international conference on Mobile computing & networking. ACM, pp. 203-205, Sep. 2013.

[4] S. Y. Wi, J. S. Park, D. W. Lee, and H. C. Yu, "A resource supplying technique for fault tolerance in hybrid cloud computing", Proceedings of KIIT Summer Conference, pp. 243-248, May 2011.

[5] S. K. Choi, K. S. Chung, and H. C. Yu, "Fault-tolerance scheduling for service sharing in mobile social cloud computing", Journal of KIIT, Vol. 11, No. 1, pp. 143-150, Jan. 2013.

[6] Y. J. Lee, "Developing Cloud-based Mobile Mashups Using Open APIs", Journal of KIIT, Vol. 12, No. 3, pp. 155-161, Mar. 2014.

[7] E. Cuervo, A. Balasubramanian, D. K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload", in Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (Mobisys), pp. 49-62, Jun. 2010.

[8] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud", Proceedings of the sixth conference on computer systems, pp. 301-314, Apr. 2011.

[9] W. Zhang, Y. Wen, K. Guan, D. Kilper, H. Luo, and D. Wu, "Energy-optimal mobile cloud computing under stochastic wireless channel", IEEE Transactions on Wireless Communications, Vol. 12, No. 9, pp. 4569-4581, Sep. 2013.

[10] Kumar, Karthik, and Yung-Hsiang Lu, "Cloud computing for mobile users: Can offloading computation save energy?", Computer, Vol. 43, No. 4, pp. 51-56, Apr. 2010.

[11] 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", INFOCOM, 2012 Proceedings IEEE, pp. 945-953, May 2012.

[12] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications", Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag New York, Inc., p. 5, Dec. 2009.

[13] Hsing-Yu Chen, Yue-Hsun Lin, and Chen-Mou Cheng, "COCA: Computation Offload to Clouds using AOP", In 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, pp. 466-473, Sep. 2012.

[14] JesusFreke, "smali/baksmali", http://baksmali.com. [Accessed: Oct. 30. 2017]

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming", ECOOP'97 – Object Oriented Programming, pp. 220-242, Jun. 1997.

[16] B. Alll and C. Tumbleson, "Dex2jar: Tools to work with android. dex and java .class files", https://github.com/pxb1988/dex2jar, [Accessed: Oct. 30. 2017]

[17] Open Handset Alliance, "Android Open Source Project", https://source.android.com, [Accessed: Oct. 30. 2017]

[18] R. L. Rivest, "Game tree searching by min/max approximation", Artificial Intelligence, Vol. 34, No. 1, pp. 77-96, Dec. 1987.

[19] wikipedia, "LINPACK benchmarks", https://en.wikipedia.org/wiki/LINPACK_benchmarks, [Accessed: Oct. 30. 2017]

저자소개

이 재 훈 (Jae-Hun Lee)

2011년 2월 : 한양대학교 컴퓨터공학과(공학사)
2013년 2월 : 한양대학교 컴퓨터소프트웨어학과(공학석사)
2013년 3월 ~ 현재 : 한양대학교 컴퓨터소프트웨어학과(박사과정)
관심분야 : 모바일 플랫폼, 분산 처리 시스템, 클라우드 컴퓨팅

강 수 용 (Soo-Yong Kang)

1996년 2월 : 서울대학교 수학과(학사)
1998년 2월 : 서울대학교 컴퓨터공학과(공학석사)
2002년 2월 : 서울대학교 컴퓨터공학과(공학박사)
2003년 ~ 현재 : 한양대학교 컴퓨터소프트웨어학과 교수
관심분야 : 분산 처리 시스템, 저장장치, 모바일 플랫폼, 클라우드 컴퓨팅, 운영체제