안드로이드 앱 리소스 캐싱 시스템과 성능분석

김태순*, 안우현*
*광운대학교 컴퓨터과학과
e-mail:coffee90506@naver.com
whahn@kw.ac.kr

Analyzing App Resource Cache in the Android Operating System

Tae Soon Kim*, Woo Hyun Ahn*
*Dept of Computer Science, Kwangwoon University

요 약

안드로이드 운영체제에서 앱과 시스템의 성능을 향상시키기 위해 효율적인 메모리 관리가 요구된다. 기존 연구는 앱 단위로 메모리를 관리하였으나 앱이 사용하는 이미지, 동영상과 같은 리소스를 저장하는 메모리 공간인 앱 리소스 캐시에 대한 분석이 없었다. 본 논문은 앱 리소스 캐시의 구조, 리소스 캐싱의 동작을 분석하고, 빈번히 접근되는 리소스의 캐싱이 앱과 시스템 성능에 어떠한 영향을 미치는지 분석한다.

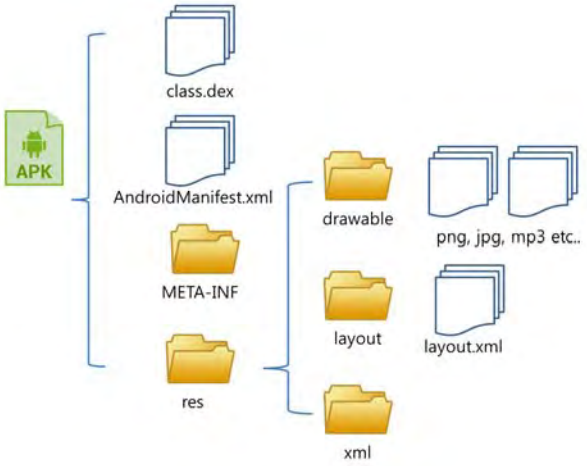
1. 서론

최근 하드웨어 기술의 발달로 스마트폰도 데스크탑 PC의 성능을 따라잡고 있다. 이에 따라 고사양을 요구하는 앱이 개발되고 스마트폰의 메모리 관리의 중요성도 증가하고 있다. 하지만 안드로이드 운영체제는 지속적인 업데이트에도 불구하고 메모리 관리에 있어 끊임없이 문제가 제기되고 있다.

안드로이드 시스템 내에는 메모리를 효율적으로 운용하기 위해 LMK(Low Memory Killer)[1] 라는 데몬이 존재한다. 안드로이드 프레임워크는 앱 프로세스 마다 우선순위를 부여한다. LMK는 이러한 우선순위를 이용하여 가용메모리가 부족할 때, 우선순위가 낮은 프로세스를 종료하여 메모리를 확보한다.

메모리 관리를 위한 기존 연구들은 다음과 같은 연구들이 있었다. 첫째, 백그라운드 프로세스의 우선순위를 임의로 조정하여 사용자가 자주 사용하는 특정 앱의 호출 성능을 향상시켰다[2]. 둘째, 앱의 사용 패턴을 예측하고, 다음에 사용되지 않을 앱을 메모리에서 제거하는 방식으로 메모리를 확보하였다 [3]. 이 연구들은 앱 프로세스 단위에서의 메모리 관리만 언급할 뿐 실제 앱의 리소스(Resource)(예로, 이미지, 동영상, UI 레이아웃)을 저장하는 메모리에 대한 자세한 분석이 없었다.

본 논문은 앱이 리소스 파일에 접근하는 I/O시간을 최소화하기 위해 리소스를 저장하는 메모리 공간인 앱 리소스 캐시를 분석한다. 이 분석을 통해 앱 단위로수행되는 리소스의 캐싱 동작을 이해하고, 빈번히 접근되는 리소스의 캐싱이 앱과 시스템 성능에 어떠한 영향을 미치는지 확인한다. 본 연구는 안드로이드 운영체제의 소스코드 분석을 통해 이루어졌으며, 리소스를 위한 효율적인 메모리 관리의 중요성을 일깨워주는 첫 번째 시도라는 점에서 큰 의미를 지닌다.



(그림 1) APK 파일의 구조

2. 리소스 캐싱 분석

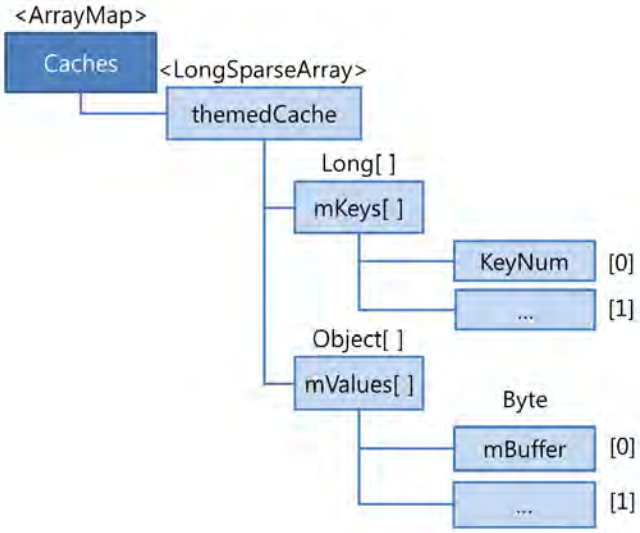
안드로이드의 메모리는 네이티브 힙(Native Heap)과 달빅 힙(Dalvik Heap) 영역 두 가지로 나뉜다. 네이티브 힙 영역은 커널에서 사용되는 메모리 영역이며 달빅 힙 영역은 달빅 가상머신이 사용하는 메모리 영역이다. 앱의 실행에 쓰이는 리소스들은 달빅 힙 영역에 할당되며, 달빅 힙 내의 메모리는 안드로이드 GC(Garbage Collector)를 통하여 관리된다[4]. GC는 앱 프로세스의 메모리가 증가할 경우 참조하고 있지 않는 객체들을 해제하여 메모리공간을 확보한다. 하지만 달빅 힙 영역은 디바이스마다 일정한 메

모리 한계가 존재한다. 따라서 앱에서 용량이 큰 리소스를 한번에 로드하는 경우 메모리 초과 오류가 발생하여 비정상 종료 될 수 있다. 구글 에서는 이러한 문제를 해결하기 위해 다량의 리소스가 사용되는 경우 LruCache 또는 DiskLruCache를 이용하여 프로그래밍 하도록 권장하고 있다.[5]

안드로이드 앱의 설치는 설치파일을 디스크에 압축해제하며 설치하지 않는다. 설치 후에도 압축형태의 파일인 APK 파일만 존재하며 APK 파일 안에는 (그림 1)과 같이 앱 실행에 필요한 리소스와 정보들이 각각의 폴더에 저장되어 있다. APK파일 안의 파일들은 크게 덱스파일, xml 파일, 리소스 파일 세 가지로 분류된다. 덱스파일은 Java 파일을 바이트코드로 변환한 파일이며, xml파일은 액티비티 레이아웃 파일들이 주를 이룬다. 마지막으로 리소스 파일은 앱에서 사용되는 사진, 음악, 영상등의 자원(Resource)파일이 해당된다.

안드로이드 앱은 실행시 필요한 리소스들을 달빅 힙 영역에 로드한다. 이때 리소스에 대한 캐싱 동작도 함께 이루어지게 된다. (그림 2)는 리소스의 캐싱이 이루어지는 과정을 간략하게 나타낸다. 첫째, 앱이 실행되면(①) 필요한 리소스를 로드하기 위해 APK파일에 접근한다(②). 둘째, 리소스 파일의 경로와 파일타입을 이용하여 고유의 키 값을 생성한다(③). 셋째, 생성된 키 값을 캐싱된 리소스의 객체와 키 값이 저장되어있는 구조체에 존재하는지 확인한다.(④). 만약 존재하지 않는 경우, 리소스를 메모리에 로드하고(⑤), 키 값과 그 객체를 구조체에 저장한 뒤 반환한다(⑥). 존재하는 경우, 키 값에 해당되는 리소스 객체의 메모리 주소를 참조하여(⑦) 그 객체를 반환한다. 넷째, 마지막으로 반환된 리소스 객체를 앱에서 사용한다(⑧).

(그림 3)은 캐싱된 리소스 객체와 키 값이 저장되는 앱 리소스 캐시의 구조를 나타낸다. 앱 리소스 캐시는 실제로 Caches라는 이름의 구조체 배열로 선언되어 있으며 내부

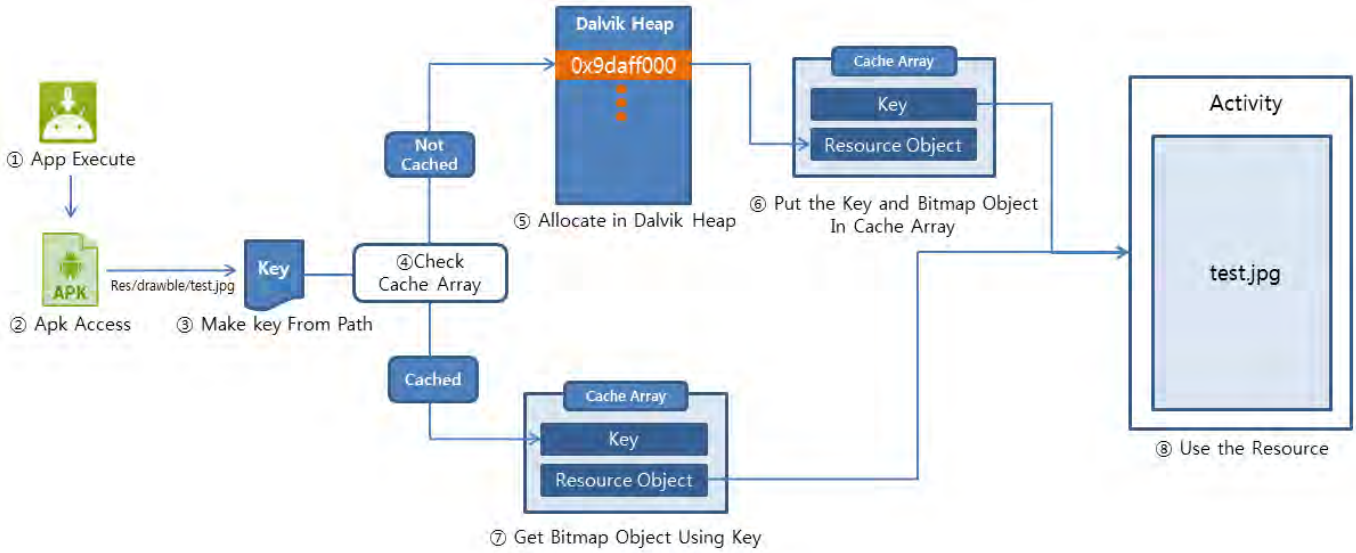


(그림 3) 앱 리소스 캐시인 Cashes구조체의 구조

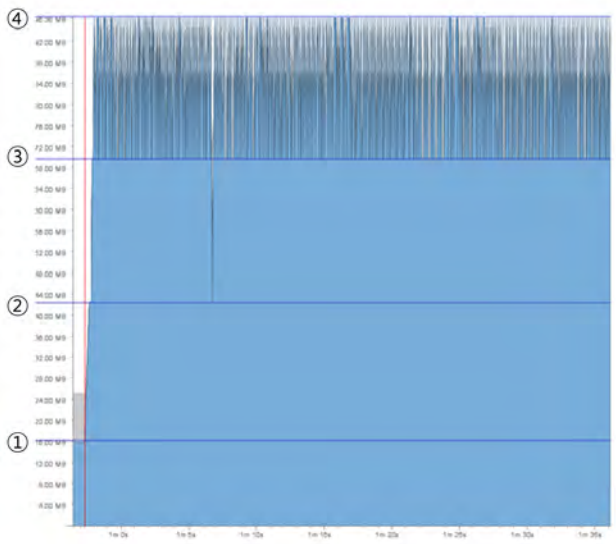
에는 themedCache라는 이름의 구조체 배열이 저장되어 있다. themedCache 구조체 배열에는 메모리에 로드된 리소스의 키 값과 메모리 주소를 가리키고 있는 리소스 객체가 저장된다. 키 값은 themedCache의 mKeys 배열에, 리소스 객체는 mValues 배열에 저장된다. 또한 mKeys배열과 mValues배열에 저장된 값은 같은 인덱스를 갖는다. 예를 들어 mKeys[0]에 저장된 키 값은 mValues[0]에 저장된 리소스 객체의 키 값이다.

3. 실험

본 논문에서는 리소스 캐싱 동작의 성능을 알아보기 위한 실험을 진행하였다. 실험을 위해 안드로이드 프레임워크에서 리소스 캐싱이 이루어지는 코드를 삭제하여 비(非)캐싱 버전을 제작하였다. 리소스에 반복적으로 접근하도록 만든 테스트앱과 구글 플레이스토어에 등록된 상용 앱을 각 버전에서 실행하여 성능을 측정하였다. 실험 기기는 안



(그림 2) 앱 리소스 캐싱 과정 모식도



(그림 4) 비캐싱 버전에서의 테스트앱 실행 메모리변화

드로이드 5.0.1 Lollipop 버전의 Nexus 5를 사용하였으며, Android Studio를 통하여 메모리 변화를 관측하였다.

3.1 테스트 앱을 통한 리소스 접근 시간 측정

첫 번째 실험으로 하나의 이미지 리소스에 100회씩 반복적으로 접근하는 테스트 앱을 제작하여 실험을 진행하였다. 실험결과 원본 버전의 실행시간은 약 235ms 이며 비캐싱 버전의 실행시간은 약 18002ms로 약 76배 정도의 차이를 보였다. 또한 비캐싱 버전의 경우 캐싱이 이루어지지 않아 지속적인 메모리 사용량의 증가가 발생한다. (그림 4)는 이런 증가를 막기 위해 실행되는 GC의 메모리 해제 동작을 보여준다. 메모리 사용량이 ④ 부분까지 증가하면 GC는 ③ 수준으로 메모리 사용량을 줄인다. 이런 GC의 동작이 매우 빈번히 발생하기 때문에 리소스 캐시가 없다면 시스템의 전반적인 성능이 저하될 수 있다.

3.2 상용 앱을 통한 앱 반응 속도 측정

두 번째 실험으로 “네이버 캘린더” 앱을 이용하여 실험을 진행하였다. “네이버 캘린더” 앱은 달력 앱으로써 각 날짜에 “스티커”라는 앱에 내장된 조그만 이미지를 달력에 첨부 할 수 있다. 총 5개의 같은 스티커와 2개의 다른 스티커를 다른 날짜에 여러개 첨부한 후 원본 버전과 비캐싱 버전에서의 앱 실행부터 로딩이 완료되는 지점까지의 시간을 측정하였다.

앱의 로딩시간을 측정하기 위해 안드로이드의 “접근성(Accessibility)” 서비스를 사용하였다. 접근성 서비스는 화면과 관련된 각종 이벤트 메시지를 제공한다[6]. 실험에서는 화면의 구성이 변화할 때 발생하는 메시지와 화면 터치할 때 발생하는 이벤트 메시지를 사용하였다. 이벤트 메시지를 통해, 앱을 실행할 때 발생하는 터치 이벤트의 시간부터 앱의 로딩이 완료되어 화면변화 이벤트 메시지가 더 이상 발생하지 않는 시점까지의 시간을 측정하였다.

실험 결과, 원본 버전은 평균 3.45초, 비캐싱 버전은 평균 4.17초의 실행시간이 소요되었다. 비캐싱 버전에서 원본 버전보다 약 21%가량의 지연시간이 발생 했다. 따라서, 비캐싱 버전에서는 중복으로 사용되는 이미지, 즉, 리소스 캐시에 중복으로 리소스를 로드하는 횟수가 증가하므로 앱 실행 시간이 증가한다.

4. 결론

본 논문에서는 앱이 사용하는 리소스의 메모리 캐싱 방법과 비캐싱 버전과 원본 버전에서의 실험을 통해 안드로이드 리소스 캐싱의 성능을 알아보았다. 그 결과 비캐싱 버전과 원본버전의 리소스 접근 시간은 약 76배, 앱 반응 속도는 약 21%의 차이가 나는 것을 확인했다. 이 정보는 리소스 캐싱에서의 메모리관리 기법을 고안하는데 이용될 수 있다.

참고문헌

[1] Developers , A. "Low RAM Configuration" URL : https://source.android.com/devices/tech/config/low-ram.html

[2] Sangseon Ryu, So-Jin Ahn, Jin-Young Choi (2014), "Improvement of the Launch Respose of an Android Application by Adjustment of LRU Algorithm based on the Low Memory Killer"

[3] Song, W., Kim, Y., Kim, H., Lim, J., & Kim, J. (2014). Personalized optimization for android smartphones. ACM Transactions on Embedded Computing Systems (TECS), 13(2s), 60.

[4] Developers, A. (2014). Managing your app's memory. URL: https://developer. android. com/training/articles/memory. html (visited on 10/28/2015).

[5] Developers, A. “Caching Bitmaps” URL : http://developer.android.com/intl/ko/training/displaying-bitmaps/cache-bitmap.html

[6] Developers, A. "Accessibility" URL : http://developer.android.com/guide/topics/ui/accessibility/index.html