Journal of Digital Contents Society Vol. 12 No. 4 Dec. 2011(pp.551 - 557)

안드로이드 2.3 달빅 가상머신에서 스택 할당 기법을 통한 메모리 성능 향상 기법

임영규*,**, 김정길***, 김신덕****

요 약

본 논문에서는 안드로이드 2.3에서 어플리케이션 실행 시 가비지 컬렉션(garbage collection)으로 인하여 발생하는 성능 저하를 감소시키기 위한 자바(Java) 객체들의 스택 할당(stack allocation) 기법을 제안하였다. 제안한 스택 할당 기법에서는 자바 객체들을 가비지 컬렉션이 되는 힙(Heap) 영역 대신에 스택에 할당함으로써 실행 시 가비지 컬렉션 대상이 되지 않게 한다. 제안한 기법의 성능 검증을 위하여 실제 자바 벤치마크에서 널리 사용되고 있는 Caffeinemark 및 자체 벤치마크 어플리케이션을 통해 안드로이드 스마트 폰에서 비교 실험을 하였다. 그 결과 자바 객체들의 스택 할당에 따르는 동작상의 오버헤드로 인한 수행 속도의 저하는 미미함을 보이면서도 가비지 컬렉션 수행 빈도는 상당히 감소시켜 어플리케이션 동작 및 사용자 인터페이스 성능 향상을 가져왔다.

Stack Allocation-based Memory Performance Improvement Technique on Android 2.3 Dalvik Virtual Machine

Yeongkyu Lim, Cheong Ghil Kim, Shin Dug Kim

Abstract

In this paper, we propose a stack allocation technique of Android Java objects in order to reduce the number of garbage collection which is one of major reason on Android performance degradation when running applications. The proposed technique is to allocate Java objects into stack rather than heap memory. To do so, stacked objects could escape the garbage collection process. We experiment the proposed technique in the latest Android 2.3 version. For the simulation, we take advantage of the well known Java benchmark, Caffeinemark, and our own. The result shows the performance degradation of Dalvik Virtual Machine execution time caused by the stack allocation of Java objects is very slight and the proposed method considerably reduces the frequency of garbage collection. This will increase application performance and give better user interfaces to Android phone users.

Keywords : Android, Garbage Collection, Stack Allocation

1. 서론

아이폰과 안드로이폰으로 촉발된 스마트 폰 시장의 폭발적인 성장은 스마트 폰을 구동시키는 핵심 요소인 운영체제(Operating System, 이하 OS))에 대한 중요성과 관심을 증대시키고 있다. 특히, 안드로이드 OS는 오픈 소스(Open Source)로 많은 제조사의 스마트 폰에 탑재되고 있다[1]. 안드로이드 OS는 iOS, Palm OS 등과 달리 대부분의 어플리케이션이 자바(Java) 언어로 개발이 되어 있으며, 가상머신(Virtual Machine. 이하 VM)에서 동작한다. 이를 위하여, 안드로이드에서는 기존에 널리 사용되고 있는 SUN의 JVM을 대신하여 리소스가 제한적인 임베디드 장비에 적합한 VM인 달빅 (Dalvik)을 채택하였다. 달빅 VM은 기존 자바 코드 대신 달빅 바이트 코드를 인터프리팅 하는 과정을 수

※ 제일저자(First Author) : 임영규
접수일:2011년 08월 25일, 수정일:2006년 09월 15일
완료일:2011년 12월 19일
* 연세대학교 컴퓨터과학과 postrain@yonsei.ac.kr
** LG전자
*** 남서울대학교 컴퓨터학과(교신저자)
(cgkim@nsu.ac.kr)
**** 연세대학교 컴퓨터과학과

행하며, 기존 자바 바이트 코드를 달빅 바이트 코드로 변환해 주는 툴로 DX(Dalvik eXchange)가 사용된다[2].

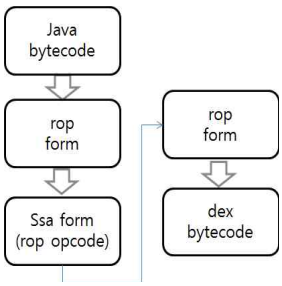
DX는 스마트 폰이나 태블릿(Tablet) PC 같은 제한된 하드웨어에서 최적화를 위한 여러 가지 기법들을 포함하고 있지만, 아직 JVM과 비교하여 개선될 부분을 많이 가지고 있다[3]. 특히, 가비지 컬렉션(garbage collection)을 줄여주기 위한 기법 중의 하나인 탈출 분석(Escape Analysis, 이하 EA)은 최신 버전인 안드로이드 2.3 진저브레드(Gingerbread)의 공식적인 릴리즈에 포함은 되지 않았지만, 소스 코드 수준에는 포함 되어 있다[4]. 진저브레드에 포함된 탈출 분석은 배열만 분석하게 되어 있으며, 해당 작업은 배열이 가지고 있는 객체 중 사용하는 객체만 변수로 선언하여, 배열의 new() 동작을 제거하는 것이다. 그러나 위 조건에 해당하는 프레임워크 코드가 많지 않기 때문에 성능 향상을 기대하기는 어렵다.

일반적으로 가비지 컬렉션이 자주 발생하고 수행 시간이 길면 안드로이드 앱에서 화면 갱신 횟수가 저하되므로 화면이 부드럽게 갱신되지 못하는 문제가 발생한다. 이 문제를 해결하기 위해 본 연구에서는 가비지 컬렉션이 관여하는 힙 메모리 활용 방식을 줄이고, 지정된 스택에 객체와 클래스를 할당하도록 해주는 스택 할당 기법을 제안하였다. 이를 통하여 가비지 컬렉션 호출 횟수를 감소시킬 수 있다[6].

본 논문의 구성은 다음과 같다. 2장에서는 달빅 VM과 DX 툴에 대한 설명을 하며, 더불어 안드로이드에서 이미 적용한 컴파일 단계에서 최적화 기법을 소개한다. 3장에서는 제안한 스택 할당 기법에 대해 설명하며, 4장에서는 제안한 기법을 안드로이드 폰에 적용하여 실험한 결과를 보인다. 마지막으로 5장에서는 실험 결과에 대한 분석과 향후 연구 방향에 대해 제시한다.

2. DX(Dalvik eXchange)

DX는 .class 파일을 달빅 VM에서 동작 가능한 .dex 파일로 변환한다. 즉, DX는 자바 바이트 코드로 구성되어 있는 .class 파일을 입력으로 받아 중간 언어인 ROP(Register OPeration)형태로 변경 후 SSA(Static Single Assignment) 형태로 변환하고 다시 ROP 형태로 변환 한다. 그리고 마지막 단계로 최종 결과물인 달빅 바이트 코드로 구성되어 있는 .dex 파일로 변환한다. 위 과정을 도식화하면 그림 1과 같다[7].



(그림 1) DX 처리 과정

DX의 모든 과정은 메소드(method) 단위로 이루어지며, 내부적으로 수행하는 변환과 최적화 역시 메소드 단위로 수행된다. SSA 형태 변환하는 것은 이후에 수행하는 최적화에 꼭 필요하거나, 향상된 결과가 나올 수 있게 하기 위함이다. ROP는 각 메소드의 제어 흐름을 기술하기 위한 중간 언어이고, SSA는 모든 변수들을 정확히 한 번만 배정한 중간 언어를 의미한다. DX의 최적화는 ROP를 SSA 형태로 바꾼 이후 실행된다.

표 1은 DX의 최적화 방식을 요약하였다. MOVE PARAM COMBINER, SCCP, LITERAL UPGRADE, CONST COLLECTOR는 DX 옵션에 따라 사용 유무를 결정 할 수 있으며, DEAD CODE REMOVER, ΦTYPE RESOLVER는 무조건적으로 실행되며, ESCAPE ANALYSIS는 코드는 일부 작성되어 있지만, 실행되지는 않는다. LITERAL UPGRADE, ESCAPE ANALYSIS, CONST COLLECTOR가 수행된 이후에는 기존에 사용되던 코드가 더 이상 사용되지 않는 경우가 발생하기 때문에 DEAD CODE REMOVER를 직접 호출한다. 최적화 과정의 의존성을 살펴보면 LITERAL_UPGRADE와 ESCAPE ANAYSIS, CONST COLLECTOR가 수행된 이후에는 DEAD CODE REMOVER가 항상 수행 되며, ΦTYPE RESOLVER는 조건 없이 항상 수행한다. 모든 메소드가 위의 과정을 마치면 DX에서 최적화 과정은 종료 된다.

DX에서 제공하는 위의 여러 가지 최적화 기법 가운데 본 연구에서는 EA 기법에 초점을 맞추었다.

<표 1> DX 최적화 요약

최적화 방식	기능
MOVE PARAM COMBINER	인자 중복 될 시 제거
SCCP	변수의 상수 여부를 체크하여 컴파일러가 직접 계산
LITERAL UPGRADE	0이나 known null을 이용하는 IF를 IF_*Z로 변경 및 Constant 인자 선언
ESCAPE ANALYSIS	객체의 유효 범위를 확인 후 힙 할당 개수를 줄임
CONST COLLECTOR	자주 사용 되는 CONST를 vRE에 할당하여 인스트럭션을 줄임
DEAD CODE REMOVER	실행되지 않는 코드 제거
ΦTYPE RESOLVER	Phi function의 경우 무조건 VOID 형으로 마킹 해놓는데 실질적인 타입을 계산

여기서 가비지 컬렉터의 동작을 줄이기 위해서는 동적 할당을 줄이는 작업이 필요한데 이는 EA를 통하여 가능하다. 진저브레드에서는 오직 배열에 대해서만 상수 치환을 수행하는데, 이의 대상이 되기 위해서는 배열의 크기가 상수가 되어야 하는 등의 여러 조건을 만족해야 한다. 실제로 진저브레드에서 EA 동작 여부를 확인하기 위하여 그림 2의 예제 코드를 만들었다.

```
int largeArray(int a, int b) {
        int array[] = new int[] {3, 4};
        int A = array[0] + a;
        int B = array[1] + b;

        if (A > B)
                return a;
        else
                return b;
}
```

(그림 2) 예제 코드

예제 코드에서 배열은 각각으로 나누어져 동작 가능하다. 이때 .dex 파일을 덤프 하여 비교 하면 아래 테이블이 보이고 있듯이 EA 분석이 이루어지기 전과 후의 효과를 확인 할 수 있다.

```
0000: const/4 v3, #int 2 // #2
0001: new-array v2, v3, [I // class@0004
0003: fill-array-data v2, 00000016
0006: const/4 v3, #int 0 // #0
0007: aget v3, v2, v3
0009: add-int v0, v3, v5
000b: const/4 v3, #int 1 // #1
000c: aget v3, v2, v3
000e: add-int v1, v3, v6
0010: if-le v0, v1, 0014 // +0004
0012: move v3, v5
0013: return v3
0014: move v3, v6
```

(그림 3) EA 분석 전

```
0000: const/4 v2, #int 3 // #3
0001: const/4 v3, #int 4 // #4
0002: add-int v0, v2, v5
0004: add-int v1, v3, v6
0006: if-le v0, v1, 000a // +0004
0008: move v2, v5
0009: return v2
000a: move v2, v6
000b: goto 0009 // -0002
```

(그림 4) EA 분석 후

그림 3과 그림 4를 비교하여 탈출 분석 수행 후를 보면 배열에 할당 되어야 할 것들이 v2, v3 레지스터(Register)에 직접 값을 기록하는 것을 볼 수 있다. 탈출 분석이 수행되기 전에는 v2 레지스터에 배열이 담기고, v2 레지스터와 인덱스를 이용하여 객체를 얻어 냈지만 탈출 분석이 수행되고 나서는 v2, v3 레지스터에 각각 이용되는 객체들을 담고 있고 배열은 사라진다. 배열을 할당하고 값을 기록하여 배열에 값을 가져오는 코드는 레지스터로 대신하고 있기 때문에 메모리 접근에 따른 오버헤드, 메모리 사용량과 배열 및 불필요한 객체의 new()를 통한 가비지 컬렉터의 호출을 줄일 수 있는 효과가 있다. 하지만 위의 테스트 코드를 포함하여 진저브레드에서 탈출 분석이 되는 대상은 작기 때문에 실제 적용을 하더라도 성능 향상은 거의 없을 것으로 예상 된다.

3. 스택 할당

모든 자바 객체는 생존범위(life scope)를 가지고 있다. 일부 객체들은 주어진 메소드 내에서만 생존하며, 또 다른 객체는 클래스 내부 또는 스레드 내에서 생존하기도 한다[5]. 자바에서의 동적 메모리 할당 기법은 프로그래머들에게 편리함을 제공하는 반면에 가비지 컬렉션을 필요로 한다. 가비지 컬렉션은 안드로이드 스마트 폰에서 사용자 인터페이스를 부드럽지 못하게 만드는 하나의 원인이다. 만약, 객체의 생존 범위가 메소드 내 또는 자식 메소드 내로 제한 된다면, 이 객체는 힙 영역 보다는 스택에 안전하게 할당 될 수 있다. 스택 할당으로 인해 가비지 컬렉터의 동작 또한 실행 횟수를 줄일 수 있다. 스택 할당은 객체 할당 시 new 동작 대신에 push 명령을 수행함으로써 객체 할당 시간을 단축시키는 장점도 있다, 또한, 객체가 메소드 내에서만 사용 되기 때문에, 메소드 수행이 끝나면 스택 포인터가 호출자의 위치로 움직이게 되고 스택에 있던 객체들은 자동적으로 제거 된다.

반면에, 힙에 할당된 객체들은 가비지 컬렉션이 끝나기 전까지는 메모리를 차지하고 있는 단점이 있다. 스택 할당의 구현은 기본적으로 배열뿐만 아니라 힙에서 객체를 할당하는 NEW_INSTANCE를 스택에서 객체를 할당하는 STACK_INSTANCE로 바꾸어 줌으로써 가능하다. 그러나 모든 객체가 스택에 할당될 수는 없는데 아래와 같은 조건의 경우엔 불가능하다.

1. 대상 객체가 타 매소드 인자로 들어간 경우
b. 대상 객체의 메소드 인자에 객체가 들어간 경우
c. 대상 객체의 리턴 값이 객체 타입인 경우
d. 대상 객체가 다른 탈출되는 객체에 지정되는 경우
e. 스레드 클래스와 같이 소스 코드만 보았을 때는 메소드 내부에서 객체를 생성하고 객체를 이용하지 않아 버리는 것 같아 보이지만 내부적으로 객체가 링크되어 있는 경우

위와 같은 경우를 피하고 객체 스택에 할당하기 위해서는 인자와 리턴 값이 자바의 프리미티브 타입만 이용 가능하며, 추적하는 객체가 타 메소드의 인자로 전달되지 않아야 한다[6].

스택 할당을 구현하기 위해서는 Dalvik 오피코드에 'STACK_INSTANCE' 라는 새로운 오피코드의 도입이 필요하다. 이는 기존 'NEW_INSTANCE' 오피코드의 스택 버전으로, 새로운 명령어를 지원하기 위해서는 달빅 VM에 많은 수정이 필요하다. 또한 DX는 정적 분석만 수행하기 때문에 일부 객체들은 스택에 할당이 될 수 있으나, 컴파일 과정에서 발견되지 않거나, 탈출하지 못하는 객체들도 있어, 이 제약을 극복하기 위하여 새로운 단계를 추가하였다. 이 단계에서는 외부 부가 기록(annotation) 파일을 참조하여 수동으로 확인된 비탈출 객체들을 annotation.csv 파일에 명시해 둔다. 구현시 DX는 위 파일에 명시된 비탈출 객체들을 NEW_INSTANCE에서 STACK_INSTANCE로 변경한다. Annotation CSV 파일의 구성은 그림 5와 같다.

```
Lkr/co/test/TestClass;.testMethod(Lkr/co/test/ArgClass;Z)V,Lkr/co/test/TargetClass;
Lkr/co/test/TestClass;.testMethod,Lkr/co/test/TargetClass;:30
Lkr/co/test/TestClass;,Lkr/co/test/TargetClass;:30
Lkr/co/test/TestClass;.testMethod,Lkr/co/test/TargetClass;
Lkr/co/test/TestClass;,Lkr/co/test/TargetClass;
Lkr/co/test/TestClass;:30
Lkr/co/test/TestClass;.testMethod(Lkr/co/test/ArgClass;Z)V
Lkr/co/test/TestClass;.testMethod
Lkr/co/test/TestClass;
```

(그림 5) Annotation CSV 파일의 구성

그림 5의 의미는 다음과 같다. kr.co.test.TestCalss의 void TestMethod(kr.co.test.ArcClass c, Boolean b) 타입의 메소드에 new()로 선언되어 있는 kr.co.test.TargetClass이며 그 중 TestClass.java의 30 라인에 선언되어 있는 new()에 대해서 NEW_INSTANCE를 STACK_INSTANCE로 교체를 한다는 의미이다. 형식은 JNI 에서 사용하는 타입 형식과 같다. class 는 'L'로 시작고 ';'로 끝나며, 패키지와 클래스를 구분하는 '.'은 '/'로 바꾸어 쓴다. 하나의 라인은 하나의 선언에 대해 지정할 수 있으며 여러 라인에 걸쳐서 끊어 쓸 수 없다. 하나의 메소드에 하나의 할당밖에 없는 경우 전체적으로 명시하지 않고 일부만 명시해도 어떤 것을 얘기하는 지 알 수 있다.

annotation.csv 파일의 위치는 빌드 명령을 수행하는 디렉터리와 같은 곳에 위치해야 하며 .dex 파일이 만들어 질 때 존재하는지 확인을 하여 해당되는 NEW_INSTANCE를 STACK_INSTANCE로 변경한다. 한번 STACK_INSTANCE로 변경된 명령은 다시 검색되지 않기 때문에 중복적으로 같은 위치를 여러 방식으로 선언을 할 수 있다.

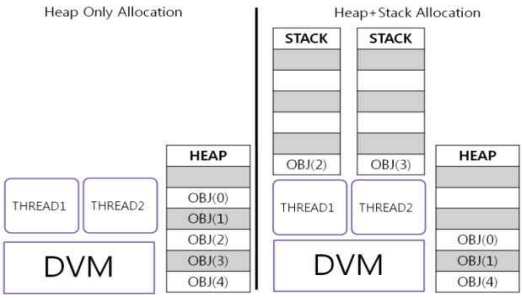
앞서 언급되었듯이 기존에 객체를 생성하고 할당할 때 NEW_INSTANCE라는 오피코드를 이용했지만, NEW_INSTANCE를 이용하면 추후 가비지 컬렉션 대상이 되기 때문에 본 연구에서는 이를 STACK_INSTANCE라는 오피코드로 변경하여 추후 가비지 컬렉션 대상이 되지 않게 하였다. STACK_INSTANCE 오피코드는 달빅 VM에서 기본적으로 지원하지 않는 오피코드여서 이를 지원하기 위해서는 달빅 VM 내부의 객체 구조, Interpreter, JIT, 가비지 컬렉션 등 많은 부분의 수정이 필요하다.

달빅 VM은 달빅 바이트 코드를 인터프리팅하여 실행하는 레지스터 기반의 인터프리터이다. 표준 JVM은 스택 기반 인터프리터로 스택에서 오퍼랜드를 읽어 이를 별도의 메모리 공간에 저장하는 방식을 취하고 있다. 그러나 달빅 VM은 레지스터 기반 인터프리터로 CPU 에서 직접 레지스터에 접근하여 명령어 디스패칭 과정을 줄여 성능 향상을 시키는 방법으로 구현되었다[3]. 스택 머신과 비교하여 불필요한 쓰기를 하지 않아도 되는 장점이 있다.

모든 달빅 객체는 첫 번째 필드로 객체 구조체를 가지고 있으며, 실행 시에 NEW-INSTANCE를 통하여 할당 받은 것인지, STACK-INSTANCE를 통하여 할당 받은 것인지에 대한 정보를 확인하기 위하여 많은 자원을 소비한다. 이를 방지하기 위하여 객체 할당 방식 확인 플래그를 추가하여 NEW_INSTANCE로 할당 받은 것인지 STACK_INSTANCE로 할당 받은 것인지 쉽게 확인할 수 있도록 하였다. 따라서 STACK_INSTANCE로 할당 받는 경우 플래그 값을 설정해 주어야 하며, 이에 따라 실행 시에 신속한 타입 확인이 가능하다.

인터프리터 스택은 달빅 VM에서 사용되는 가상의 스택이다. 모든 메소드는 인터프리터 스택 내에 자신의 스택 프레임을 가지고 있다. 스택 프레임 내부에는 입력 파라미터, 로컬 데이터, 스택 프레임 자체를 유지하기 위한 정보들과 스레드의 객체 할당 스택이 있다. 객체를 스택에 할당하기 위해서는 각각의 스레드에 객체 할당 스택이 추가 되어야 한다. 객체 할당 스택은 인터프리터 스택과는 별개의 것이며, 초기에 16KB 사이즈를 할당 받는다. 스레드가 객체를 할당하기 위하여 보다 많은 스택 영역을 요청할 경우, 달빅 VM은 16KB 단위로 40개까지 할당해준다. 시스템이 여유 공간이 없을 경우에는 STACK_INSTANCE를 NEW_INSTANCE 오피코드로 변경하여 힙에 할당 되도록 한다.

그림 6의 오른쪽 부분은 각각의 Dalvik 스레드에 대한 객체 할당 스택을 형상화 하여 보여주고 있다. 메소드가 호출되거나 복귀할 때마다 스레드의 객체 할당 스택은 증가하거나 감소를 하게 된다.



(그림 6) 객체 할당 스택

진저브레드에서 VM의 변동 사항중 하나가 병렬수행 방식의 가비지 컬렉션의 도입이다. 병렬수행 가비지 컬렉션은 힙 할당량이 일정한 크기를 넘을 경우 별도의 스레드에 의해 가비지 컬렉션을 수행하여 기존 방식에 의해 발생하는 응용 스레드가 동작 멈춤 시간을 많이 감소시켰다. 병렬수행 가비지 컬렉션이 발생하면 CardTable이 생성된다. CardTable은 가비지 컬렉션과 다른 스레드가 동시에 동작하고 있을 때 다른 스레드에 의한 객체의 수정에 대응하기 위해 수정된 지역의 메모리(128 바이트 단위)를 마킹하여 병렬수행 가비지 컬렉터가 다시 독점적으로 수행하는 시점에서 마킹된 메모리 영역에 속하는 모든 객체를 다시 스캔함으로 새롭게 생성된 객체가 사라지지 않는 현상을 방지하기 위한 장치

이다.

스택 할당 방식에서는 객체 할당 스택에서 객체를 마킹하기 위하여 하나의 단계가 더 추가된다. 비록 가비지 컬렉터에 의한 객체 할당 스택의 추가 스캔 작업을 필요로 하지만, 스캔이 되어야 할 총 객체의 수는 스택 할당으로 인하여 증가하지는 않으므로 객체 할당 스택으로 인한 성능 저하는 무시할 수 있다. 그림 7의 (a)는 기존 가비지 컬렉션의 동작 과정이며, (b)는 스택 할당을 추가한 가비지 컬렉션 동작 과정이다.

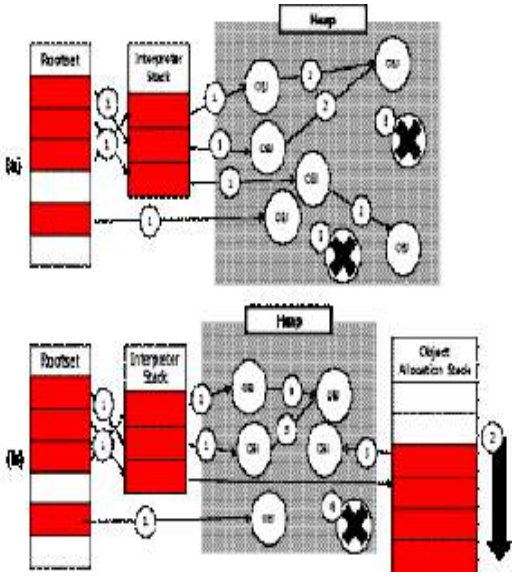


그림 7 가비지 컬렉션 수행 과정

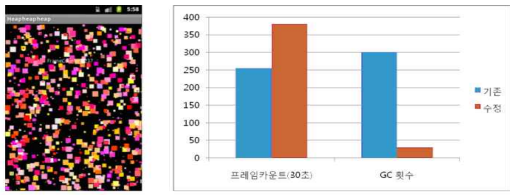
4. 실험결과

객체의 할당을 힙에서 스택으로 변경 할 경우 여러 가지 장점이 존재한다. 스택 할당은 new()를 사용하지 않으므로 메모리 할당에 따른 오버헤드가 감소하며, 결과적으로 메모리 사용량도 감소하여 가비지 컬렉터 수행 횟수 또한 감소된다. 그리고 객체가 더 이상 사용되지 않을 경우 기존의 스캔이나 스윕 과정 필요 없이 해당 객체를 소유한 메소드가 반환되면 해당 스택에서 자동 제거된다.

정확한 성능 평가를 위하여 최신 버전인 안드로이드 2.3이 장착된 ARM11 600Mhz CPU 성능의 스마트 폰에서 달빅 VM 코드 및 DX Tool 소스를 직접 수정하여 실험하였다. 먼저, Caffeinemark 벤치마크를 수행하였다[8]. 그림 8은 실험 결과를 보여주고 있다. 우측이 스택 할당 방식을 적용한 결과인데 스택 할당에 따른 전체 달빅 VM의 성능 차이는 거의 없다. Method score와 같이 메소드 호출에 오버헤드가 추가되었기 때문에 테스트 결과는 조금 낮게 나오지만, 수행되는 동안 가비지 컬렉션의 전체적인 감소로 인해 실행 시간이 달라지지 않기 때문에 총점에는 거의 변화가 없다.

<표 2> Caffeinemark 수행 결과

Score	Conventional	Proposed
Sleve	748(98)	776(98)
Loop	3128(2017)	3166(2017)
Logic	3553(0)	3498(0)
String	1575(708)	1494(708)
Float	1303(185)	1298(185)
Method	1362(166650)	1427(166650)
Overall	1689	1695



(그림 8) 샘플 프로그램 이미지 실험 결과

또한, 가비지 컬렉션 측면에서 성능 평가를 위해 테스트 어플리케이션을 제작하였는데, 그림 9의 왼쪽에서 볼 수 있듯이 임의의 위치에서 임의 크기의 박스를 그리며 많은 객체 할당을 수행한다. 이러한 할당 가운데 대다수는 STACK-INSTANCE로 대체가 가능하다. 실험 결과는 그림 5의 오른쪽과 같다. 30초간 수행 결과에서 프레임 수는 기존 방식 255개에서 수정 방식 380개로 더 많은 프레임을 그릴 수 있었으며, 가비지 컬렉션 횟수도 기존 301회에서 30회로 크게 감소되었다.

5. 결론

본 논문에서는 안드로이드 달빅 VM의 가비지 컬렉션 발생 빈도 감소를 통한 으용 성능 향상을 위하여 컴파일 시점에 적용할 수 있는 스택 할당 기법을 제안하였다. Caffeinemark 벤치마크를 통한 실험 결과 스택 할당으로 인한 오버헤드가 작음을 알 수 있으며, 가비지 컬렉션 성능 평가를 통해 객체 할당이 많은 어플리케이션의 경우 가비지 컬렉션 발생 빈도 감소를 통해 UI 프레임 수의 증대가 가능하였다. 향후에는 달빅 VM 내부의 가비지 컬렉션 알고리즘 개선을 통해 실행 시에 가비지 컬렉션 발생 빈도 및 수행 시간을 감소시킬 수 있는 방법에 대해 연구할 계획이다.

참 고 문 헌

[1] http://news.cnet.com/8301-13506_3-20051610-17.html

[2] Dalvik Virtual Machine. http://www.dalvikvm.com.

[3] Technical Report, “Analysis of Dalvik Virtual Machine and Class Path Library,” http://imsciences.edu.pk/serg/wp-content/uploads/2009/07/Analysis-of-Dalvik-VM.pdf.

[4] Escapee Analysis, http://www.java2s.com/Open-Source/Android/android-core/platform-dalvik/com/android/dx/ssa/EscapeAnalysis.java.java-doc.htm

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar and Sam Midkiff, “Escape Analysis on Java,” ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications(OOPSLA99). Denver, Colorado, November 1, 1999 pp. 1-19.

[6] David Gay and Bjarne Steensgaard, “Stack allocating objects in Java,” Research Report, Microsoft Research, 1999.

[7] Dan Bornstein, “Dalvik virtual machine: internals,”http://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf“.

[8] Caffeinemark: http://www.benchmarkhq.ru/cm30

임 영 규

1999년 : 고려대 컴퓨터학과(석사)
2009년 : 연세대 컴퓨터과학과 (박사수료)

1999년~현 재: LG전자 책임연구원
2005년~현 재: 연세대 컴퓨터과학과 박사과정
관심분야 : Mobile Computing, Mobile Multimedia

김정길

2003년 : 연세대학교 대학원 (공학석사)
2006년 : 연세대학교 대학원 (공학박사)

2006년~2007년 : 연세대학교 컴퓨터과학과 PostDoc,
2007년~2008년 : 연세대학교 컴퓨터과학과 연구교수
2008년~현 재 : 남서울대학교 컴퓨터학과 조교수
관심분야 : 멀티미디어 임베디드 시스템, 모바일 컴퓨팅

김 신 덕

1981년 : 연세대학교 전자공학과 (공학사)
1987년 : Univ. of Oklahoma EE (공학석사)
1991년 : Purdue University, CE and EE (공학박사)

1995년~현 재: 연세대학교 컴퓨터과학과 교수
관심분야 : 차세대 메모리, 모바일 웹 브라우저, 유비쿼터스 컴퓨팅