DBpia

Method Signature를 이용한 안드로이드 런타임 성능 향상

정회원 조 인 휘*°, 김 원 일**

Enhancement of Android Runtime Performance using Method Signature

Inwhee Joe*°, Wonil Kim** *Regular Members*

요 약

본 논문에서는 Android Dalvik 가상머신의 trace JIT 구현 코드를 수정하여 특정 Java 메소드를 별도의 프로파일링 과정 없이 바로 JIT 컴파일이 이루어지도록 구현하였다. 동일한 Java 메소드에 대해 원본 Dalvik 가상머신과 본 논문의 방식을 적용한 가상머신에서 각각 실행하여 수행 시간을 측정하였고 본 논문이 제안한 방식을 이용하면 약 30% 정도의 성능 향상을 가져올 수 있음을 확인하였다. 본 논문에서는 Android Dalvik의 구조를 살펴보아 Dalvik 가상머신에 대한 이해를 돕고 변경한 Dalvik 가상머신 부분을 코드 수준에서 자세히 설명한다. 산업체에서는 Dalvik 가상머신의 수행 성능 최적화와 같은 Android의 핵심 기술 부분을 향상하여 Android를 사용하고 있는 타 경쟁사와 차별점을 가진 제품을 만드는데 도움이 될 수 있을 것이다.

Key Words : Android, Dalvik, Virtual Machine, JIT, Compiler

ABSTRACT

In this paper, we have shown Dalvik virtual machine implementation to reduce the profiling overhead from the trace of the JIT compiler for the specific method. By running the same Java method on the original Dalvik VM and the modified Dalvik VM, we have achieved around 30% performance improvement with this algorithm. In this paper, to increase the reader's understanding of Android Dalvik virtual machine, we will explain the architecture of Dalvik JIT compiler and we will provide the detailed explanation with source codes for modified parts of Dalvik virtual machine. From the industry perspective, we can expect competitive benefits over the competitors with performance improvement in Android core.

Ⅰ. 서 론

애플의 iPhone으로 시작된 스마트폰 열풍이 전 세계적으로 뜨겁다. 기존 휴대전화와 달리 스마트폰의 경우 어떤 운영체제를 사용하는지가 중요한 제품 선택 기준이 되고 있다. 특히, 구글이 오픈소스로 내놓은 Android의 경우 현재 iPhone에 대적할 수 있는 유일한 솔루션으로 인식되고 있다.

Android를 운영체제로 사용한 제품들은 S/W 측면으로는 서로 큰 차이점이 없어 제품의 차별화가 어렵다. Android의 핵심 기술인 Dalvik 가상머신의 성능을 최적화할 수 있다면 성능 및 기능을 향상시킬 수 있어 경쟁력을 높일 수 있을 것이다.

Android의 Dalvik Virtual Machine은 DEX라는 고

* 한양대학교 컴퓨터공학부 이동네트워크 연구실(iwjoe@hanyang.ac.kr), (° : 교신저자)
** Nokia Siemens Networks 스마트랩(wonil.kim@gmail.com)
논문번호 : KICS2011-04-187, 접수일자 : 2011년 4월 14일, 최종논문접수일자 : 2011년 11월 9일

www.dbpia.co.kr

유의 바이너리 파일 형식을 사용한다. DEX 형식의 프로그램을 특정 H/W에서 실행하기 위해서 Dalvik 가상머신은 DEX 명령어를 실시간에 CPU가 인식할 수 있는 기계어 명령어로 해석(인터프리터)한다. 일반적인 프로그램을 인터프리터 기반의 가상머신에서 실행하는 경우 미리 컴파일 된 프로그램을 실행하는 것보다 대략 10배 정도 성능이 느리다[2].

이러한 가상머신의 성능 문제를 해결하기 위해 다양한 연구가 진행되었다. DIR(Directly Interpretable Representation) [3]을 직접 해석하여 처리할 수 있는 특별한 H/W를 사용하여 성능을 향상하는 방법[4] 부터 멀티쓰레드를 활용한 인터프리터 성능 향상[5] 등의 여러 연구 결과가 발표되었다.

가상머신의 인터프리터 성능을 향상시키기 위한 여러 연구가 진행되었지만 최적화된 컴파일러를 사용하는 경우의 성능에 필적할만한 수행 성능 향상은 이루지 못하였다[6]. 이러한 문제를 해결하기 위해 실행시간 컴파일러를 이용한 성능 향상 연구가 이루어지기 시작하였으며 현재는 JVM(Java Virtual Machine), CLR(Common Language Runtime)과 같은 상용 가상머신에서 동적 실시간 컴파일러를 채택하고 있다. Android 2.2 버전인 Froyo 부터는 실시간 컴파일러를 제공하여 Android 프로그램의 수행 성능을 향상시켰다[1].

[7]의 연구결과를 보면 응용프로그램의 수행 성능에 영향을 미치는 부분은 일부의 반복되는 코드에 의한 영향이 대부분이다. 이러한 사실에 근거하여 Dalvik 가상머신은 for, while 문과 같이 많은 반복되는 코드만 선택하여 컴파일을 수행한다.

실시간 컴파일러가 코드를 선택적으로 컴파일하기 위해서는 일반적으로 아래와 같은 3가지 단계를 필요로 한다[6].

1. 최적화가 필요한 코드를 찾기 위한 프로파일링
2. 코드 중 실제 최적화를 적용할 부분을 결정
3. 최종 선택된 코드 부분을 동적으로 컴파일

위 단계들은 가상머신의 수행성능에 오버헤드로 작용하게 된다.

본 논문이 제안하는 성능 향상 방법은 Java 메소드의 이름을 미리 정의한 형태로 선언한 경우에는 최적화 후보 코드 선정 및 결정 단계를 생략하고 바로 실시간 컴파일을 실행하도록 하여 1, 2 단계의 오버헤드를 줄이는 방식이다.

2장에서는 Dalvik JIT 컴파일러의 문제점에 대해 살펴본다. 3장에서는 Dalvik의 인터프리터, 컴파일러 등의 코드를 수정하여 성능을 향상시키는 방법에 대해 살펴본다. 4장에서는 변경된 Dalvik 가상머신의 성능을 테스트 프로그램을 이용하여 검증하고 Dalvik의 수행 성능이 개선되었음을 확인한다.

Ⅱ. 관련 연구

JIT 컴파일러의 도입으로 Android 런타임의 성능이 많이 향상되었으나 아래와 같은 프로파일링 오버헤드 문제점이 있다.

Dalvik은 반복되는 코드를 찾기 위해 후위 분기가 발생하는 코드 지점마다 아래와 같은 프로파일링 과정을 수행한다.

1. JIT Profiling Table의 포인터를 얻는다.
2. PC 값에 대한 hash를 계산한다.
3. hash 값을 이용하여 JIT Profiling Table의 요소 값을 얻는다.
4. 이 값을 1 만큼 감소시킨 후 0이 되는지 확인한다.
5. 0으로 감소한 경우면 Compile Selection Mode로 진입한다.
6. 그렇지 않으면 DEX 바이트 코드를 인터프리트 한다.

위 과정을 처리하기위해 Dalvik 가상머신 인터프리터에 구현된 프로파일링 코드를 살펴보면 아래와 같다.

```
GET_JIT_PROF_TABLE(r0)

common_updateProfile:

eor  r3,rPC,rPC,lsr #12 @cheap,but fast hash
                          @function.
lsl  r3,r3,#(32 - JIT_PROF_SIZE_LOG_2)
ldrb r1,[r0,r3,lsr #(32 - JIT_PROF_SIZE_LOG_2)]
GET_INST_OPCODE(ip)
subs   r1,r1,#1        @ decrement counter
strb   r1,[r0,r3,lsr #(32- JIT_PROF_SIZE_LOG_2)]
GOTO_OPCODE_IFNE(ip)
```

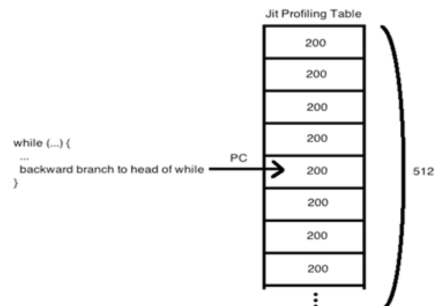

그림 1. JIT Profiling Table 배열의 구조

www.dbpia.co.kr

위와 같은 프로파일링 코드가 반복문의 후위 분기 시 마다 추가적으로 수행된다.

Ⅲ. 제안하는 성능 개선 방안

3.1 Method Signature 기반의 JIT 컴파일

앞에서 살펴본 Dalvik의 오버헤드를 개선하기 위해 제안하는 방법은 다음과 같은 조건을 만족해야 한다.

1. 자주 실행되는 코드 블록을 찾기 위한 코드 오버헤드를 제거해야 한다.
2. 컴파일이 필요한 코드를 선택하는 과정에 의한 오버헤드를 제거해야 한다.
3. 메소드 단위의 컴파일이 가능해야 한다.

위와 같은 조건을 만족하기 위해 본 논문에서 제안하는 해결책은 메소드의 이름을 이용한 아이디어로 Dalvik 가상머신에 큰 변경 없이도 쉽게 구현이 가능하다는 장점이 있다. 그림 2는 아이디어를 구현하기 위한 대략적인 처리 순서를 보여주고 있다.

1. 응용프로그램 개발자가 작성한 코드를 외부 프로파일링 도구를 사용하여 가장 성능에 영향을 미치는 메소드를 찾는다.
2. 해당 메소드의 이름을 미리 정의한 형태로 메소드 이름을 변경한다. 성능을 향상시킬 메소드에 특정 문자열 signature를 붙인 형태가 된다.
3. Android SDK가 제공하는 DEX 변환 도구에서 __ct를 이름 후위에 가진 메소드를 발견하면 해당 메소드에 ACC_COMPILE _TARGET 이라는 access flag를 지정한다.
4. Dalvik 가상머신에서 메소드를 호출 시 ACC_COMPILE_TARGET access fla 값이 설정되어 있는 경우에는 프로파일링 과정을 생략하고 바로 JIT 컴파일 한다.
5. 컴파일 된 결과를 실행한다.

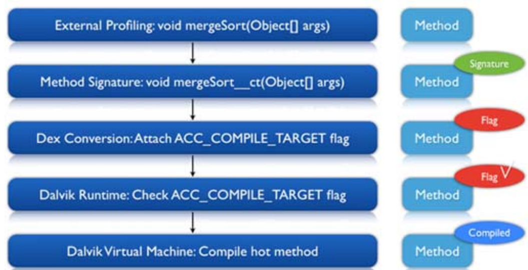

그림 2. 메소드 이름을 이용한 개선 방안

이 방법을 따르면 프로그램 수행 성능에 중요한 특정 메소드들을 기존 Dalvik의 실시간 컴파일 과정에서 필요한 프로파일링 및 컴파일 대상 코드 선택 오버헤드 없이 JIT 컴파일을 수행할 수 있어 Dalvik의 수행 성능을 향상시킬 수 있다. 본 논문의 개발 및 실험 환경은 다음과 같다.

Android 버전: Android 2.2 Froyo
호스트 운영체제: Ubuntu Linux 10.10
컴파일러: GCC v4.4.5, JDK v5.0
Android 실행 환경: Android SDK QEMU ARMv5 CPU 에뮬레이터
디버거: ADB, GDB, DDMS

3.2 구현을 위한 Dalvik 소스 코드 수정

3.2.1 DEX 변환 도구 수정

Dalvik 가상머신이 실행시간에 __ct 문자열을 이름 후위에 가진 메소드를 찾기 위해 DEX 변환 도구를 수정한다. Dalvik 가상머신을 직접 수정하여 메소드 이름을 비교하게 구현할 수도 있지만 이 방법은 모든 메소드가 호출 될 때마다 메소드의 이름 문자열을 비교해야하는 오버헤드가 발생할 수 있어 DEX 변환도구를 수정하는 방법을 택하였다.

Android 오픈소스 파일 중 CfTranslator.java 파일을 보면 Java 메소드 바이트 코드를 DEX 포맷으로 변경 처리하는 processMethods 메소드가 구현되어 있다. processMethods 메소드를 수정하여 __ct 후위 문자열을 가진 Java 메소드를 찾아 컴파일이 필요한 메소드임을 나타내는 ACC_COMPILE _TARGET access flag를 지정한다.

```
private static void processMethods
    (DirectClassFile cf,CfOptions args,
    ClassDefItem out)
{
 MethodList methods = cf.getMethods();
 int sz = methods.size();

 for (int i = 0; i < sz; i++) {
  Method one = methods.get(i);
  ...
  String methodName =  one.getName().getString();
  if (methodName.endsWith("__ct")) {
   accessFlags |= AccessFlags.ACC_COMPILE_TARGET;
  }
   ...
 }
}
```

Android는 설치하는 DEX 파일의 포맷에 문제가

www.dbpia.co.kr

없는지 검증을 하는데 기존에는 없던 ACC_COMPILE_TARGET access flag 값을 가진 메소드가 있기 때문에 검증 과정에서 실패하고 설치가 이루어지지 않는다. DEX의 메소드 정보가 올바른지 확인하는 과정은 DexSwapVerify.c 파일의 verifyMethods 함수에서 이루어진다.

```
if (((accessFlags & ~ACC_METHOD_MASK) != 0)
   || (isSynchronized && !allowSynchronized))

LOGE("Bogus method access flags %x @ %d\n",
accessFlags, i);
 return false;
}
```

이 문제를 해결하기 위해서 ACC_METHOD_MASK의 상수 값에 새로이 정의한 ACC_COMPILE_TARGET을 추가해준다.

3.2.2 Dalvik 인터프리터 변경

Dalvik 가상머신이 DEX 코드를 실행할 때 ACC_COMPILE_TARGET이 지정된 메소드의 경우에는 실시간 컴파일을 바로 수행하도록 하기 위해서는 AMRv5용 인터프리터 소스 코드를 수정해주어야 한다.

위 소스 코드를 보면 r3 레지스터에 현재 호출하려는 메소드의 access flag 값을 저장 후 ACC_COMPILE_TARGET 값과 비교한다. 만일 ACC_COMPILE_TARGET 값이 access flag에 설정되어 있는 경우라면 .LinvokeCompileTarget 라벨로 지정한 어셈블러로 분기한다.

```
#if   defined(WITH_JIT)
    ldr       r3, [r0, #offMethod_accessFlags]
    tst       r3, #ACC_COMPILE_TARGET
    GET_JIT_PROF_TABLE(r0)
    mov       rFP, r1
    bne       .LinvokeCompileTarget
    GET_PREFETCHED_OPCODE(ip, r9)
    mov       rINST, r9
    str       rFP, [r2, #offThread_curFrame]
    cmp       r0,#0
    bne       common_updateProfile
    GOTO_OPCODE(ip)   @ jump to next instruction

.LinvokeCompileTarget:
    GET_PREFETCHED_OPCODE(ip, r9)
    mov       rINST, r9
    str       rFP, [r2, #offThread_curFrame]
    cmp       r0,#0
    bne       common_methodProfile
    GOTO_OPCODE(ip)
#else
    ...
```

.LinvokeCompileTarget 라벨의 어셈블러 코드는 common_methodprofile로 분기하여 현재 호출하려는 메소드를 Dalvik 컴파일러 쓰레드에서 컴파일 하도록 처리 한다.

호출하려는 메소드가 이미 JIT 컴파일된 결과가 Translation Cache에 저장되어 있는지 확인 후 그렇지 않다면 메소드 단위의 컴파일을 요청하기 위해 kJitTMethodRequest 값을 인자로 하여 common_selectTrace로 분기하여 Dalvik 가상머신의 동작 모드를 컴파일 할 메소드 선택 모드로 변경한다.

Jit.c 파일에 구현된 dvmCheckJit 함수를 수정하여 common_selectTrace에 의해 지정된 trace 선택 모드를 확인 후 메소드 컴파일 요청이라면 dvmCompileWorkEnqueue 함수를 호출하여 현재 메소드를 컴파일러 큐에 등록하여 Dalvik의 컴파일러 쓰레드가 해당 메소드의 DEX 코드를 기계어로 컴파일 한다.

Dalvik의 JIT 컴파일러 과정은 컴파일러 쓰레드에 의해 처리된다. 이 쓰레드는 자신이 관리하는 큐에 컴파일이 필요한 코드 블록이 있으면 깨어나 그 코드 블록을 컴파일하고 다시 큐에 컴파일 요청이 들어올 때를 기다린다.

지금까지 설명한 것처럼 Dalvik의 인터프리터의 어셈블러, C 소스 코드를 수정하여 __ct를 후위문자열로 가진 메소드는 컴파일러 쓰레드가 관리하는 큐에 등록하여 쓰레드가 깨어날 때 메소드에 대한 컴파일이 이루어지도록 수정하였다. 다음으로 구현해야 할 소스 코드 부분은 메소드를 JIT 컴파일 하는 과정이다.

3.2.3 Method 컴파일러 변경

앞에서 언급한 것처럼 Android 2.2 버전의 Dalvik 소스는 메소드 단위의 컴파일을 지원하지 않는다.

하지만, 이 논문의 아이디어를 실현하기 위해서는 꼭 메소드 단위의 컴파일이 가능해야 하기 때문에 dvmCompileMethod 함수를 수정하여 메소드 단위 컴파일이 가능하도록 하였다.

dvmCompileMethod 함수의 실행 과정 및 디버깅 중 문제점을 수정한 부분에 대해서 살펴보자.

1. 메소드에 대한 DEX 명령어의 시작부터 끝까지 순차적으로 진행하면서 하나의 기본 블록에 모두 추가한다. 동시에 기본 블록의 경계를 찾는다.
2. 1번 과정에서 찾아진 블록의 경계에 따라 하나의 기본 블록을 여러 개의 기본 블록으로 나눈다. 나누어진 기본 블록들은 컴파일러가 분석하기 쉬운 특성을 가진 단위를 의미한다. 컴파일러가 분석을 쉽게 하기 위해 각 기본 블록이 하나의 진입점과 하나의 반환점을 가지도록 만든다. 즉, 하나의 진입점에 해당하는 코드 부분을 제외하고는 다른 코드 부분은 분기 명령어의 대상이 되지 않는다[8].
3. 2번 과정에 의해 나누어진 기본 블록 전체를 순차

www.dbpia.co.kr

조회하여 관계가 있으면 기본 블록들을 서로 연결해준다. 블록 코드에서 전위 혹은 후위 분기를 하는 경우를 확인한 후 분기의 대상이 되는 명령어를 진입점으로 가지는 기본 블록이 있는지 확인하여 해당 블록이 있다면 연결하여 준다.

4. 정리된 기본 블록 정보를 dvmCompilerMIR2LIR 함수를 호출하여 MIR을 LIR로 바꾼다.
5. LIR을 dvmCompilerAssembleLIR 함수를 호출하여 CPU에서 실행 가능한 기계어로 어셈블 한다.

1, 2, 3 번 과정에 의해 컴파일을 위해 필요한 기본 블록에 대한 정보가 모두 정리되었으면 아래 코드와 같이 (4, 5번 과정) Dalvik이 제공하는 JIT 컴파일 관련 함수들을 호출하여 기계어 코드를 생성할 수 있다. dvmCompilerAssembleLIR 메소드로 생성된 기계어 코드는 Dalvik의 Translation Cache 메모리에 저장되어 메소드 호출 시 해당 기계어 코드가 실행된다.

```
cUnit.instructionSet =
dvmCompilerInstructionSet();

dvmCompilerMIR2LIR(&cUnit);

dvmCompilerAssembleLIR(&cUnit, info);

/* just for debugging */
dvmCompilerDumpCompilationUnit(&cUnit);

dvmCompilerArenaReset();
```

Ⅳ. 실험 결과

Dalvik VM을 수정한 결과 성능이 실제로 향상되는지 살펴보기 위해 아래와 같은 조건으로 실험을 진행하였다.

1. ARMv5의 Trace JIT 방식은 반복문이 200번 이상 수행된 경우만 골라서 JIT 컴파일을 하기 때문에 반복문의 횟수를 200번을 경계로 비교하면 성능 향상의 결과를 쉽게 확인할 수 있을 것이다.
2. 복잡한 부동소수점 연산을 수행하는 메소드를 작성한 후 약 400회 정도 반복문에서 수행에 걸린 시간을 측정한다. 각각 변경이 없는 Dalvik 가상머신과 메소드 이름을 이용한 JIT 컴파일 방법을 적용하여 수정한 Dalvik 가상머신에서 시간을 측정하여 메소드 이름에 의한 방법이 어느 정도 성능을 향상시킬 수 있는지 검토한다.

실행 시간 측정을 위해 부동소수점 연산을 400회 반복하도록 하였다. 테스트 코드를 메소드 이름 방식의 구현을 적용한 Dalvik VM과 변경하지 않은 Dalvik VM으로 15회씩 실행하여 평균 수행 시간을 측정하였다.

그림 3에서 볼 수 있듯이 메소드 이름을 이용한 Dalvik 수정을 적용한 Modified JIT 컴파일러 방식이 평균 30% 정도의 수행 성능이 향상된 결과를 볼 수 있다.

반복문의 횟수가 더 커질수록 초기 200번 동안 발생하는 오버헤드의 비율이 작아지기 때문에 Modified JIT에 의한 성능 개선 효과가 상대적으로 작아질 수 있다. 이를 확인해보기 위해 반복문의 횟수를 800회로 증가하여 동일한 테스트를 수행해보았다.

그림 4의 성능 측정치를 보면 Original JIT와 Modified JIT의 수행 성능이 약 10% 정도 차이를 보이고 있다. 기존 400번 실행한 경우보다는 800번 실행한 경우가 성능향상의 차이가 크지 않음을 알 수 있다.

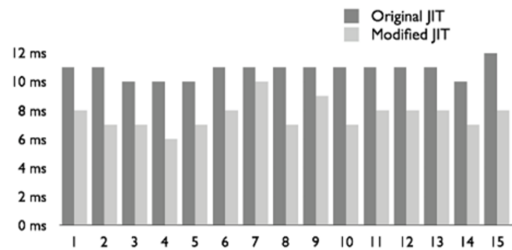

그림 3. 실험 결과 1

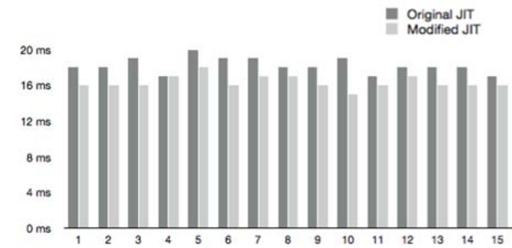

그림 4. 실험 결과 2

Ⅴ. 결 론

Dalvik의 Trace 기반 JIT 컴파일러는 자주 실행되는 코드 부분을 찾기 위해 후위분기가 발생하는 코드마다 프로파일링 오버헤드가 추가된다.

본 논문에서는 미리 정의한 형태의 메소드 이름를 이용하여 Dalvik의 JIT 컴파일러가 가진 프로파일링 오버헤드를 제거하도록 가상머신을 수정하여 응용프

www.dbpia.co.kr

로그램 코드의 수행 성능을 향상 시킬 수 있음을 확인하였다.

참 고 문 헌

〔1〕 A JIT Compiler for Android's Dalvik VM - *Google IO 2010*

〔2〕 A code compression system based on pipelined interpreters – *Software Practice and Experience* Sept. 1999

〔3〕 Levels of representation of programs and the architectur of universal host machines - Proc. 11th Annu. *Workshop Microprogramming*, 1978

〔4〕 Design of a LISP-based microprocessor – *Communication of ACM* vol. 23

〔5〕 Threaded code – Communication of ACM, vol. 16

〔6〕 A Survey of Adaptive Optimization in Virtual Machines – *Proceedings of the IEEE*, vol. 93

〔7〕 An empirical study of FORTRAN programs - *Softw.Pract. Exper*., 1971, vol. 1, pp. 105-133.

〔8〕 GNU Compiler Collection Internals – *15.1 Basic Block*

조 인 휘 (Inwhee Joe) 정회원

1983년 2월 한양대학교 전자공학과
1994년 12월 미국 University of Arizona, Electrical and Computer Engineering, M.S.
1998년 9월 미국 Georgia Tech, Electrical and Computer Engineering, Ph.D.
1992년 12월 (주) 데이콤 종합연구소 선임연구원
2000년 6월 미국 Oak Ridge 국립연구소 연구원
2002년 8월 미국 Bellcore Lab (Telcordia) 연구원
2002년 9월~현재 한양대학교 컴퓨터공학부 교수
<관심분야> Mobile Internet, Cellular System and PCS, Sensor Networks, 임베디드 시스템

김 원 일 (Wonil Kim) 정회원

1996년 2월 대진대학교 전자공학과 학사
2011년 2월 한양대학교 컴퓨터공학부 석사
1999년 8월 Microsoft, PSS
2005년 1월 Sun Microsystems, Java 연구소
2011년 4월 Nokia Siemens Networks, 스마트랩
<관심분야> Virtual Machine, LTE, Mobile Software Platform, 임베디드 시스템

www.dbpia.co.kr