자바 바이트코드를 GVM 코드로 번역하기 위한 언어적 특성 처리

고 영관[O+], 고 석훈[++], 오 세만[+]
동국대학교 컴퓨터공학과[+], (주)신지소프트 기술이사[++]
{capsbada, smoh}@dgu.ac.kr[+], shko@sinjisoft.com[++]

Handling the Language Dependent Features for Translating Java Bytecode to GVM Code

Young-Koan Ko[O+], Seok-Hoon Ko[++], Se-Man Oh[+]
Dept. of Computer Engineering, Dongguk University[+]
Research & Development Center of SINJISOFT Co., Ltd.[++]

요 약

휴대용 무선기기의 성능 향상과 동적인 응용프로그램 실행의 필요성에 따라 모바일 응용프로그램은 기존의 네이티브 애플리케이션 제작 방법에서 가상기계(Virtual Machine)를 탑재하여 실행하는 가상기계 애플리케이션 제작 방법으로 변화하였다. 가상기계를 이용한 애플리케이션의 실행은 플랫폼 독립적인 실행이 가능하며 또한 효과적인 다운로드 솔루션을 통한 동적인 응용프로그램의 실행이 가능하다. 이러한 배경으로 KVM과 GVM(General Virtual Machine) 등의 가상기계가 출현하였다. 그러나 각 플랫폼에 맞는 애플리케이션은 서로 호환되지 않으므로 가상기계 언어간의 번역을 통해 목적 가상기계에서의 실행이 요구되며 자바 바이트코드를 순수 국내 기술로 개발된 GVM 코드로 번역하는 것이 바람직하다.

본 논문에서는 KVM에서 실행가능한 자바 바이트코드를 GVM에서 실행하기 위해 바이트코드를 GVM 코드인 SAL(Sinji Assembly Language)로 번역하는 번역기를 설계하고 구현한다. 이를 위해 자바의 언어 독립적 특성 외에 언어 의존적인 특성을 처리하기 위한 방법을 제안하며 언어 의존적인 특성을 배열과 문자열, 클래스와 객체, 예외, 스레드로 분류하여 처리한다.

1. 서 론

응용프로그램의 개발 방법 및 실행 방법은 크게 네이티브 애플리케이션과 가상기계 애플리케이션으로 나눌 수 있으며 전자는 이제까지 사용했던 방법으로 실행속도 면에서는 탁월한 장점을 갖는다. 그러나, 플랫폼이 바뀌면 모든 응용프로그램을 변경해야 할뿐만 아니라 심지어는 사용할 수 없게 된다. 이러한 단점을 극복하기 위해서 가상기계를 탑재하여 응용프로그램을 실행시켜주는 가상기계 솔루션이 등장하였으며 특히, 휴대폰 환경에서는 프로세서 및 운영체제의 다양성과 잦은 변경으로 인하여 가상기계를 이용하는 것이 적합하고 더욱이 다운로드 솔루션에서는 가상기계 애플리케이션이 타당한 방법이다.

이와 같은 장점으로 인하여 현재 무선 플랫폼 시장에서는 KVM(Kilo Virtual Machine)과 GVM(General Virtual Machine)이 사용되고 있으며 동적인 응용 프로그램을 효과적으로 실행시켜주고 있다. 그러나, 한 응용프로그램이 서로 다른 가상기계에서 실행되지 않으므로 가상기계간 호환성이 요구된다. KVM은 자바를 기본 언어로 갖는 가상기계로서 Sun Microsystems사에서 제안한 기술이지만, GVM은 순수 국내 기술로 개발된 가상기계이다. 따라서, KVM 코드를 GVM 코드로 번역하여 자바로 작성된 응용프로그램을 GVM에서 실행시켜 주는 것이 필요하다.

본 논문에서는 KVM 코드를 GVM 코드로 번역하기 위한 번역기를 설계하고 구현하였으며 번역 과정은 크게 언어 독립적인 부분과 언어 의존적인 부분으로 나누어 처리하였다. 언어 독립적인 부분은 일반적인 연산 코드에 해당되며 대부분 일대일 매핑으로 번역할 수 있으나 언어 의존적인 부분은 자바 언어가 갖는 특성으로 인해 단순 매핑 방법으로 처리할 수 없다. 따라서 본 논문에서는 언어 의존적인 특성을 배열과 문자열 처리, 클래스와 객체 처리, 예외 처리, 스레드 처리 등으로 분류하여 체계적으로 처리한다.

본 논문에서 구현한 번역기를 사용하면 GVM의 기본

본 연구는 한국과학재단의 특정기초연구(과제번호:1999-1-30300-3) 지원에 의한 것임.

언어인 MobileC로 작성된 응용프로그램뿐만 아니라 J2ME/MIDP로 작성된 자바 응용프로그램도 GVM에서 실행할 수 있게 된다.

2. 관련 연구

KVM(Kilo Virtual Machine)은 적은 메모리 공간과 낮은 데이터 전송률의 특징을 갖는 휴대용 무선기기에 적합하도록 JVM을 축소한 형태이며 JVM의 특징인 플랫폼 독립성과 스레드, 예외 처리, 가비지 콜렉션 등의 특징을 제한적으로 갖고 있다.

KVM은 부동소수점 연산을 지원하지 않으므로 J2SE의 231개 바이트코드 중 [그림 1]과 같이 49개가 제외된다[1].

- fconst_0, fconst_1, fconst_2, dconst_0, dconst_1
- fload, fload_n, dload, dload_n
- fstore, fstore_n, dstore, dstore_n
- faload, daload, fastore, dastore
- newarray T_FLOAT, newarray T_DOUBLE
- fadd, dadd, fsub, dsub, fmul, dmul, fdiv, ddiv
- frem, drem, fneg, dneg, fcmpl, fcmpg, dcmpl, dcmpg
- i2f, f2i, i2d, d2i, l2f, f2l, l2d, d2l, f2d, d2f
- freturn, dreturn

[그림 1] J2ME에서 제외된 바이트코드

GVM(General Virtual Machine)은 휴대폰 환경에 적합하도록 설계되었으며 MobileC를 기본언어로 사용한다. GVM의 실행방식은 Event-Driven 방식으로 처리하는 특징을 갖으며 각각의 이벤트 핸들러 함수를 사용하여 응용프로그램을 쉽게 구현할 수 있는 장점이 있다.

자바 언어에 의존적인 특성이 없는 일반적인 연산 코드는 [그림 2]와 같이 일대일 매핑 테이블을 사용하여 의미가 동등한 GVM 코드로 처리한다.

```
// 스택에 상수 값을 저장
iconst_arg
{
    pushc arg
}

// 스택의 값을 지역변수에 저장
istore_arg
{
    popz arg
}
```

```
// 스택에 지역변수의 값 저장
iload_arg
{
    pushz arg
}
// 스택의 값을 비교 후 분기
if_icmple_arg
{
    le
    tjp arg
}
```

[그림 2] 일대일 매핑 테이블

3. 언어 의존적 특성

자바는 객체지향, 스레드, 예외처리 등의 언어적 특성을 갖고 있다. 자바 바이트코드를 GVM 코드로 변환하는데 있어 이러한 언어적 특성을 GVM에서 수행할 수 있도록 처리해야 한다. 자바 언어 의존적인 특성의 내용과 처리 방법은 다음과 같다.

3.1 배열과 문자열 처리

자바에서 배열은 newarray, anewarray, multianewarray와 new 코드를 통해 객체를 생성하며 가상기계는 객체의 크기만큼 힙에 공간을 할당한 후 참조 주소를 반환한다[2,3,4]. 반환된 주소는 배열의 형태에 따라 baload, caload, saload, iaload, laload, aaload 등의 코드로서 스택에 객체의 참조 주소를 저장한다[2,3,4]. 문자열의 처리는 new를 통한 문자열 객체를 생성하며 문자열 상수는 가상기계의 상수 풀에 저장된다.

3.2 클래스와 객체 처리

클래스는 멤버변수와 메소드로 구성되어 있다. 멤버변수는 getstatic, putstatic, getfield, putfield 등의 코드를 통해 접근되며 메소드의 호출은 invokespecial, invokevirtual 등의 코드를 통해 수행된다[3]. 이러한 멤버변수와 메소드는 클래스의 계층적인 구조에 따라 메모리 상의 위치와 해당 코드의 실행주소가 결정되므로 가상기계가 계층적인 구조와 클래스로 생성된 객체의 공간에 대해 처리할 수 있도록 클래스 파일(*.class)로부터 클래스 정보를 추출하여 SAL로 표현한다.

3.3 예외 처리

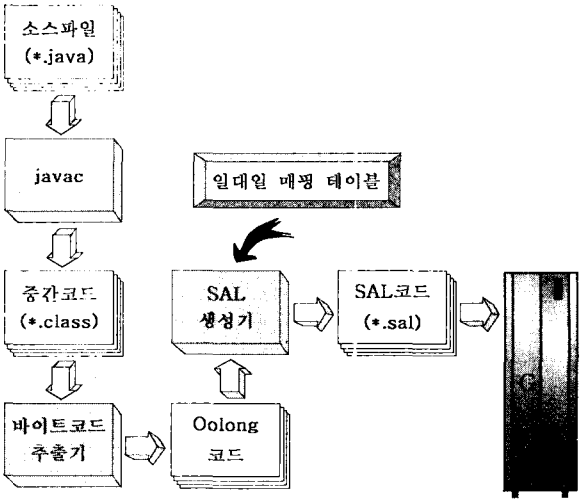
자바에서 try-catch-finally 문으로 구성되는 예외 구문은 J2ME에서는 구문의 복잡성과 수행 능력의 과부하로 인하여 finally 문은 지원하지 않는다[1]. 따라서, try-catch 문에 의한 바이트코드는 각 블록의 시작점과 끝점에 대한 분기문의 형태로 번역되며 가상기계로 시작 주소, 끝 주소, 목표 주소와 예외 형태의 정보로 구성되는 예외 테이블을 전달한다. 가상기계에선 예외 테이블을 주시하며 해당 구역 내에서 예외가 발생되는지 판단하며 발생되는 예외의 형태에 따라 목표 주소로 분기할 수 있도록 처리한다.

3.4 스레드 처리

스레드와 관련한 바이트코드는 동기화를 위한 monitorenter와 monitorexit가 있으며, 해당구역의 접근 방지와 해제의 역할을 한다. 따라서, monitorenter에 의한 해당 구역의 접근 방지 시 다른 스레드가 해당 구역의 코드에 접근을 시도하면 예외로 간주하여 구역 밖으로 분기할 수 있도록 예외 테이블을 구성한다.

4. 번역기의 설계 및 구현

바이트코드를 SAL로 번역하는 번역기의 시스템 구성도는 [그림 3]과 같다.



[그림 3] 시스템 구성도

본 시스템은 자바로 작성된 소스 파일(*.java)을 자바 컴파일러(javac)를 통해 중간언어인 클래스파일(*.class)을 얻는다. 그리고, 클래스파일을 바이트코드 추출기를 통해 어셈블리 형태의 Oolong 코드로 변환한 후 SAL 생성기의 입력으로 한다. SAL 생성기는 자바 언어 독립적인 코드에 대해 일대일 매핑 테이블을 사용하여 번역을 하며 언어 의존적인 코드를 가상기계에서 실행할 수 있도록 클래스 정보와 예외 테이블을 포함한 SAL 파일을 생성한다. 생성된 SAL 파일은 목적 가상기계인 GVM에서 실행된다.

5. 실험결과

자바 소스 파일인 Test.java를 javac의 입력으로 하여 클래스 파일을 생성한다. 바이트코드 추출기는 생성된 클래스 파일을 입력으로 하여 바이트코드를 Oolong 구문의 어셈블리 형태로 추출하며 SAL 생성기는 추출된 어셈블리 형태의 바이트코드를 입력받아 [그림 4]와 같이 목적 코드인 SAL을 생성한다.

```
// 어셈블리 형태의 바이트코드
I0:   iconst_2
I1:   istore_1
I2:   goto I46
I5:   iconst_0
I6:   istore 5
        (중략)
I45:  istore_1
I46:  iload_1
I47:  sipush 500
I50:  if_icmple I5
I53:  return
```

```
// 번역된 SAL
main:   pushc   2
        popz    1
        ujp     $$1:
$$0:    pushc   0
        popz    5
        (중략)
        popz    1
$$1:    pushz   1
        pushw   500
        le
        tjp     1
$$2:    end
```

[그림 4] 바이트코드에서 SAL로의 변환

6. 결론 및 향후 연구

자바 바이트코드를 GVM 코드로 번역하기 위해, 언어 독립적인 부분과 언어 의존적인 부분으로 나누어 설계하고 구현하였으며 특히, 본 논문에서는 언어 의존적인 특성을 고려하였다.

이와 같이 구현된 번역기는 GVM의 기본언어인 MobileC 뿐만 아니라 자바 언어를 사용하여 제작한 모바일 응용프로그램을 GVM에서 실행이 가능하게 하였으며 가상기계를 통한 응용프로그램의 실행을 더욱 융통성 있게 하였다.

앞으로 다양한 언어에 융합할 수 있도록 GVM을 개선해야하며 가상기계의 표준화 작업이 필요하다. 또한, 새로운 가상기계의 코드를 표준화된 GVM 코드로 번역하는 작업이 연구되어야 할 것이다.

참고문헌

[1] CLDC Specification v1.0, Sun Microsystems, http://java.sun.com/products/cldc

[2] Bill Venners, Inside the Java Virtual Machine, McGraw-Hill, 1998.

[3] Joshua Engel, Programming for the Java Virtual Machine, Addison Wesley, 2000.

[4] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, Addison Wesley, 2000.