정보학연구 제4권 제1호 2001년 3월(pp.1-7)
The Journal of Information Technology. Vol.4. No.1

자바 바이트코드로부터 JNI를 사용한 C 코드의 변환

권혜은*/김상훈**

요 약

자바 프로그램은 플랫폼 독립적이라는 장점을 갖는 반면에 그의 실행이 가상 기계를 통하여 이루어지기 때문에 실행 시간의 비효율성을 가진다. 이러한 문제를 극복하기 위해 just-in-time(JIT) 컴파일러, 오프라인 바이트코드 컴파일러와 같은 다양한 해결 방법이 제안되어 왔다. 그러나 JIT 컴파일러는 실행시간에 바이트코드로부터 네이티브 코드로의 번역이 일어나므로 실행시간 부담을 가진다. 그리고 순수 오프라인 바이트코드 컴파일러는 동적 클래스 적재(dynamic class loading)의 어려움을 가진다. 본 논문에서는 동적으로 바이트코드를 적재할 수 있는 능력을 유지하면서, JIT 보다 더 효율적 실행이 가능한 방법을 제안한다. 또한 기존의 bytecode-to-C 번역기와는 달리, 우리의 번역기는 자바 네이티브 인터페이스(JNI)를 사용함으로서 JDK 실행 환경과의 완벽한 호환성을 유지한다. 본 연구의 결과로 바이트코드를 JNI를 사용한 C 코드로 변환하기 위한 번역기를 설계하고 구현하였다.

Ⅰ. 서론

플랫폼에 독립적인 자바프로그램은 실행속도가 저하되는 심각한 문제점을 갖는다. 이러한 문제점을 해결하기 위해 도입된 기술이 JIT 컴파일러(Just In Time Compiler)와 오프라인 바이트코드 컴파일러(offline bytecode compiler)이다.[1] JIT 컴파일러는 바이트코드가 실행되기 전에 번역되기 때문에 실행되지 않는 코드의 번역은 피할 수 있지만 실행시간에 바이트코드를 컴파일하고 최적화 하는 부담을 갖는다.[2] 순수 오프라인 바이트코드 컴파일러는 프로그램 실행 전에 코드를 번역하므로 실행시간의 부담을 줄일 수 있지만, 자바의 동적 클래스 적재(dynamic class loading)를 지원하지 못한다.

이러한 문제점을 해결하기 위해 제안된 방법이 바이트코드를 C 코드로 변환하여 실행시키는 방법이다. 이와 관련된 기존 연구에는 Harissa[1], Toba[3], TurboJ[4], Jolt[5] 등이 있다. 이들은 순수 오프라인 컴파일러 형태로 구성되어 동적 클래스 적재를 지원하지 못한다든지 또는 JDK release 1.0의 NMI(Native Method Interface)를 사용하고 있어 현재 Java 2 SDK release 1.2를 지원하고 있지 못한다는 문제점을 가진다.

본 논문에서는 자바의 각 메소드들을 네이티브 메소드로 변환하여 향상된 실행속도를 가질 수 있는 변환기(이하 MyJNItool)를 설계하고 구현하였다. 네이티브 메소드를 생성을 위한 C 프로그램은 표준화된 JNI(Java Native Interface) 기술을 사용하여 전형적인 오프라인 바이트코드 컴파일러와는 달리 자바가상머신에 독립적이며, 동적 클래스 적재를 지원한다. 또한 표준화된

* 세명대학교 대학원 전산정보학과
** 세명대학교 소프트웨어학과 조교수

JNI를 사용하고 있어 현재의 JDK와 완벽한 호환성를 가진다.

II. 관련연구

2.1. 제안된 바이트코드 컴파일러

기존의 바이트코드 컴파일러에는 Harissa[1], Toba[3], TurboJ[4], Jolt[5]등 다양한 시스템이 있다. Harissa와 Toba는 클래스파일을 C 프로그램으로만 변환한다. Harissa는 JDK1.0.2에서 동작하며 동적 클래스적재는 지원하지만, 스레드를 지원하지 않고, Toba는 JDK1.1에서 동작하며 스레드를 지원하지만, 클래스의 동적 적재를 지원하지 않는다. TurboJ와 Jolt는 스텁 자바프로그램(stub java program)과 C 프로그램을 생성한다. Jolt는 NMI(Native Method Interface)를 사용함으로써 자바가상머신에 의존적으로 구현되었으며, 오버로드된 메소드나 생성자를 번역할 수 없다. Jolt에 의해 번역된 클래스를 사용하기 위해서는 호출자가 직접 공유라이브러리를 적재해야 하는 문제점이 있다.

2.2. 바이트코드의 실행

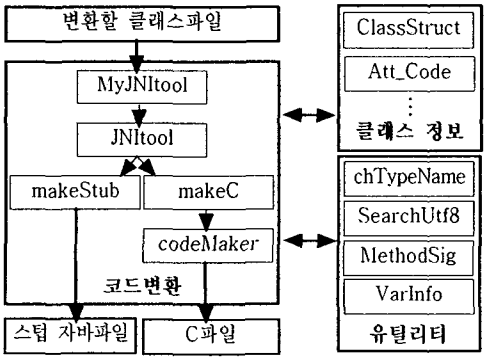
자바가상머신은 프로그램을 실행하기 위해 각 스레드마다 하나의 자바가상머신 스택을 생성한다. 자바가상머신 스택은 각 메소드 호출 시 지역변수 배열과 피 연산자 스택을 갖는 새로운 프레임(Frame)을 생성한다. 지역변수 배열은 연산의 임시결과와 메소드 호출 시 전달받은 매개변수가 저장되고, 피 연산자 스택은 연산결과, 상수, 메소드 호출 시 전달할 매개변수 등이 저장된다. 메소드가 종료되면 프레임은 제거된다.[6]

III. MyJNItool의 구조

MyJNItool은 클래스파일을 입력으로 받아 스텁 자바프로그램과 C 프로그램을 생성한다. 클래스파일을 이용해서 출력 프로그램을 생성함으로써, 변환할 프로그램의 소스프로그램이 필요하지 않으며 올바른 구문과 의미를 가지는 것이 보장된다. 또한 기존의 번역기가 NMI를 사용한 것과 달리 JNI를 이용하여 자바가상머신의 구현이 달라지더라도 이미 생성된 라이브러리의 이용이 가능하다. 또한 오버로드된 메소드, 생성자, 정적 초기화문의 변환이 가능하다.

3.1. 전체구조

MyJNItool은 (그림 1)과 같이 하나의 클래스파일이 입력되면, 클래스파일을 분석하고 정보를 얻어 스텁 자바프로그램과 바이트코드를 변환한 C프로그램이 생성된다.

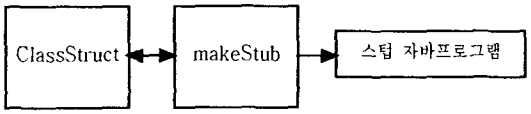


(그림 1) MyJNItool의 구조

MyJNItool 클래스는 프로그램을 시작할 때 변환할 파일이 주어졌는지 확인하여 JNItool 클래스를 생성한다. JNItool 클래스는 주어진 파일에서 자바가상머신 명세서의 클래스파일 구조에 따라 주어진 파일을 분석하여 ClassStruct 클래스를 생성하고, makeStub 클래스와 makeC 클래스를 이용하여 스텁 자바프로그램과 C 프로그램을 생성한다. Util 패키지에 포함되는 chTypeName, SearchUtf8, MethodSig 클래스들은 생성해 둔 클래스파일 정보를 제공한다. chTypeName 클래스는 클래스파일, C파일, myJNItool에서 사용되는 데이터 타입과 타입정보간의 변환을 담당한다. SearchUtf8은 상수 풀의 인덱스를 사용하여 클래스 이름, 메소드 이름, 필드이름, 디스크립터 등의 정보를 얻는데 사용된다. MethodSig 클래스는 디스크립터들을 C 형태로 변환하고, codeMaker에서 사용될 지역변수 배열의 초기 값을 결정한다.

3.2. makeStub 클래스

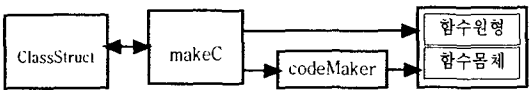
makeStub 클래스는 변환할 클래스에 관한 정보를 가지고 있는 ClassStruct 클래스에서 정보를 얻어 스텁 자바프로그램을 생성한다. 클래스파일 내의 정적 초기화문은 MyJNItool에 의해 변환된 이름으로 C 파일에 구현되고, 스텁 자바프로그램에서 정적 초기화문을 이용해 해당 네이티브 메소드를 호출한다. 생성자 또한, 이름을 변경하여 C 파일에 기록한다. 스텁 자바프로그램에서 생성자는 C 프로그램 내에서 변경된 이름을 가지는 생성자를 호출한다.



(그림 2). makeStub 클래스의 기본구조

3.3. makeC 클래스

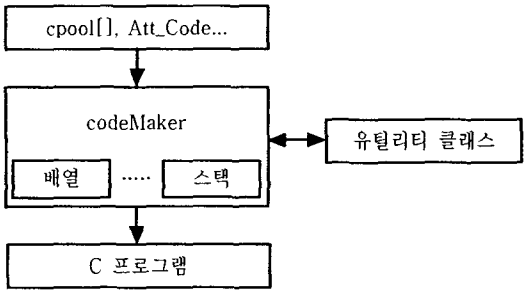
makeC 클래스는 주어진 클래스의 모든 메소드의 이름을 자바 네이티브 인터페이스 함수 이름 규칙에 맞도록 변환한 후 바이트코드 분석에 필요한 정보와 함께 codeMaker 클래스를 생성한다. codeMaker 클래스는 주어진 정보에 의해 바이트코드를 분석하면서 C 프로그램을 생성한다.



(그림 3) makeC 클래스의 기본구조

3.4. codeMaker 클래스

codeMaker 클래스는 생성 시에 주어진 정보에 의해 C 프로그램을 생성한다. C 프로그램 생성에는 codeMaker 클래스 내에 선언된 배열과 스택을 이용한다. 배열과 스택은 자바 가상머신의 지역변수 배열과 피 연산자 스택과 같은 방법으로 값이 결정되면서, JNI를 이용한 C 프로그램이 생성된다.

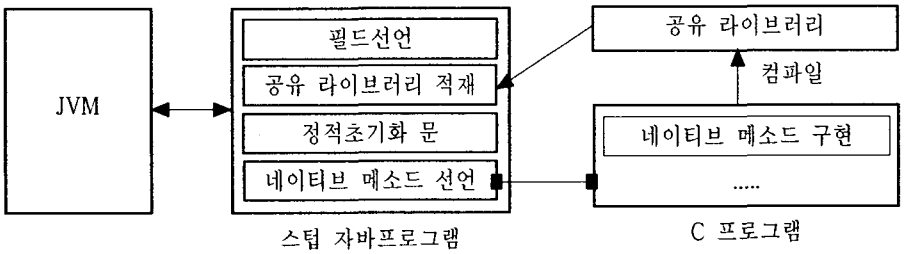


(그림 4) codeMaker 클래스의 기본구조

3.5. 번역된 결과의 실행형태

MyJNItool은 스텁 자바프로그램과 C 프로그

램을 생성한다. 스텁 자바프로그램은 필드선언, 공유라이브러리 적재, 정적초기화 문, 네이티브 메소드 선언문으로 구성되어 있다. 이는 자바 컴파일러에 의해 컴파일 되고 자바가상머신을 통해 실행된다. C 프로그램에는 스텁 자바프로그램에서 선언된 네이티브 메소드의 구현부분이 있다. 이는 C 컴파일러에 의해 컴파일 되고 최적화된 공유 라이브러리로 생성되어 스텁 자바프로그램에 의해 적재되고 자바가상머신에 의해 사용된다.



(그림 5) MyJNItool에 의해 생성된 프로그램의 실행형태

Ⅳ. 실험 및 결과

4.1. 실험 환경

실험 환경은 Intel Pentium 233MHz의 CPU와 64MB RAM의 하드웨어에 운영체제는 RedHat Linux 6.0을 사용하였고, 자바는 JDK1.3.0, C 컴파일러는 gcc를 이용하였다. JIT컴파일러는 실행시간에 컴파일과 최적화에 대한 부담을 갖기 때문에 MyJNItool에 의해 생성된 프로그램보다 실행속도가 떨어진다. 루프의 횟수 체크, 피보나치 수열, 버블정렬, 완전 수를 구하는 예제를 통하여 이를 확인하였다.

4.2. 예제 프로그램과 실행 결과

완전수를 구하는 소스 프로그램에 대한 MyJNItool의 변환 결과는 표와 같다. <표 1>은 MyJNItool로의 입력 프로그램으로 완전수를 구하는 예제이다. <표 2>와 <표 3>은 MyJNItool의 출력인 스텁 자바프로그램과 JNI를 사용한 C 프로그램이다.

〈표 2〉 입력 프로그램

```
public class Perfect {
  public static void main(String args[]) {
        int end = Integer.parseInt(args[0]);
        print(end);
  }
  public static void print(int end)  {
        int i, j, tmp;
        boolean flag = true;

  System.out.print("1, 2");
        for(i = 3; i < end; i++) {
            for(j = 2; j < i; j++) {
                  tmp = i % j;
                  if( tmp == 0 ) flag = false;
            }
            if(flag) System.out.print(", " + i);
            flag = true;
        }
    }
}
```

〈표 3〉 스텁 자바프로그램

```
public class  Perfect extends java.lang.Object {
    static {        System.loadLibrary("Perfect");        }

    public  Perfect ()  { myJNItoolinit(); }
    native public  void  myJNItoolinit () ;

    native public static  void  main (java.lang.String[] var0) ;
    native public static  void  print (int var0) ;
}
```

V. 결론 및 향후 연구 과제

자바 플랫폼에 종속적이거나 실행시간이 중요한 프로그램을 작성하려면, 프로그래머는 자바 이외의 언어에 대한 사전지식이 필요하다. 본 연구의 결과로 만들어진 MyJNItool은 바이트코드를 스텁 자바프로그램과 JNI를 이용한 C 프

〈표 4〉 JNI를 사용한 C 프로그램

```
#include <jni.h>

void JNICALL Java_Perfect_myJNItoolinit (JNIEnv *env, jobject obj) {
jobject var2;
jclass cls3;
jmethodID mid4;

label_1 :   var2 = obj;
label_2 :   cls3 = (*env)->FindClass(env, "java/lang/Object");
   mid4 = (*env)->GetMethodID(env, cls3, "<init>", "()V");
   (*env)->CallNonvirtualVoidMethod(env, var2, cls3, mid4);
label_5 :   /* return */  return ;
}

void JNICALL Java_Perfect_main (JNIEnv *env, jclass clz, jobjectArray localv0) {
                                                    ....
label_9 :   cls10 = (*env)->FindClass(env, "Perfect");
   mid11 = (*env)->GetStaticMethodID(env, cls10, "print", "(I)V");
   (*env)->CallStaticVoidMethod(env, cls10, mid11, var9);
label_12 :   return ;
}

void JNICALL Java_Perfect_print (JNIEnv *env, jclass clz, jint localv0) {
                                                    ...
label_13 :   localv1 = var14;
label_14 :   goto label_77;
label_18 :   localv2 = var15;
label_19 :   goto label_36;
label_22 :   var16 = localv1;
label_23 :   var17 = localv2;
label_24 :   var18 = var16 % var17;
label_25 :   localv3 = var18;
label_26 :   var19 = localv3;
label_27 :   if(var19 != 0)  goto label_33;
label_31 :   localv4 = var20;
label_33 :   localv2 += 1;
                                                    ...
label_68 :   cls43 = (*env)->FindClass(env, "java/io/PrintStream");
   mid44 = (*env)->GetMethodID(env, cls43, "print", "(Ljava/lang/String;)V");
   (*env)->CallNonvirtualVoidMethod(env, var26, cls43, mid44, var42);
                                                    ....
}
```

로그램으로 번역하고, 혼합형 실행(mixed-mode execution)기법을 사용하여 동적으로 바이트코드를 적재할 수 있는 능력을 유지하면서, JIT 컴파일러 보다 더 효율적인 실행속도를 가진다. 또한, 생성되는 C 프로그램은 자바 네이티브 인터페이스를 사용함으로써 이를 지원하는 모든 자바가상머신에서 실행될 수 있으며, 네이티브 메소드를 작성하는 경우 헤더 파일 생성 도구의 사용과 매개변수로 전달되는 객체에 대한 처리 부담을 줄일 수 있는 부가적인 효과도 가진다.

네이티브 메소드 내에서 시간 부담이 큰 클래스를 생성하여 사용하는 경우 프로그래머의 의도에 따라 이를 추적하여 변환하는 기능과 네이티브 메소드 작성을 위한 도구로 사용되는 경우에 이를 지원하기 위한 GUI환경의 구현을 향후 연구과제로 하여 보완할 것이다.

참고 문헌

[1] Gilled Muller, Ulrik Pagh Schultz, 'Harissa: A Hybrid Approach to Java Execution' IEEE Software, V.16 N.2, 44-+, 1999

[2] OpenJIT homepage http://www.openjit.org

[3] Toba homepage http://www.cs.arizona.edu/sumatra/toba, 1998

[4] TurboJ Documentation http://hpk.felk.cvnt.cz/turboj/contents.html

[5] Jolt Homepage, http://www.sbktech.org/jolt.html, 1996

[6] Tim Lindholm, Frank Yellin, 'The Java Virtual Machine Specification Second Edition', Addison Wesley, 1999

[7] Sheng Liang, 'The Java Native Interface Programmer's Guide and Specification', Addison Wesley, 1999

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, 'The Java Language Specification', Addison Wesley, 1999

[9] Bill Venners, Inside the JAVA 2 Virtual Machine, Mc Graw Hill, 1999

[10] Joshua Engel, Programming for the Java Virtual Machine, Addison Wesley

Translation of Java Bytecode into C code with the JNI

Hye-Eun, Kwon*/Sang-Hoon, Kim**

Abstract

The well-known tradeoff of Java's portability is the inefficiency of its basic execution model, which relies on the interpretation of an virtual machine. Many solutions have been proposed to overcome this problem, such as just-in-time(JIT) and offline bytecode compilers. However, JIT compiler can not avoid the overhead of runtime. since it translate bytecode into native code at runtime. And, pure offline bytecode compiler limits the ability of dynamic class loading. In this paper, we present an approach which preserves the ability to dynamically load bytecode, and is more efficient than JIT. In contrast to existing bytecode-to-C translator using the old NMI, our translator maintain complete compatibility and portability through using the Java Native Interface(JNI) standard. We have designed and implemented an translator for converting bytecode to C code with JNI. named MyJNItool.

* Dept. of computer & information, Semyung Univ.
** Dept. of software, Semyung Univ.