정보학연구 제5권 제4호 2002년 12월(pp.35-44)
The Journal of Information Technology, Vol.5, No.4

JNI를 사용한 혼합형 실행 환경

김상훈*

요 약

자바의 이전성은 해석 방식에 따른 비효율성을 낳는다. 또한 일반 응용 프로그램에서 자바의 표준 클래스 라이브러리는 플랫폼 의존적인 기능을 제공하지 못한다. 이러한 여러 가지 문제를 해결하기 위해 JNI, JIT, 오프라인 바이트코드 컴파일러가 제안되어왔다. 본 논문에서는 자바 가상기계가 네이티브 코드와 함께 실행되는 혼합형 실행 모델을 제시한다. 현 실행 모델과 번역기는 동적으로 바이트코드를 적재할 수 있는 능력이 주어지며 JNI의 수월한 사용을 제공한다. 본 시스템은 JIT 보다 효율적이며, JNI의 개념이 부족한 프로그래머가 쉽게 네이티브코드 작성할 수 있도록 한다.

Ⅰ. 서론

썬 마이크로시스템(Sun MicroSystems)에서 개발한 자바는 다양한 플랫폼에서 실행 가능한 코드를 지원하는 객체지향언어이다. 번역된 바이트코드는 다양한 하드웨어 환경에서 재 컴파일 과정 없이 실행 가능하므로 플랫폼 독립적인 언어라 한다. 이러한 자바 언어는 기계 독립적인 특징인 높은 이식성을 가지는 반면에 소프트웨어적으로 구축된 자바 가상기계에서 처리되므로 느린 실행 속도와 하드웨어 의존적인 처리의 어려움이라는 문제점을 가진다. 느린 실행 속도란 문제점은 실행시간 컴파일러인 JIT(Just In Time Compiler)에 의해 보완되며, 하드웨어 의존적인 처리는 자바 네이티브 인터페이스(JNI: Java Native Interface)에 의해 일반적으로 이루어진다.

* 세명대학교 소프트웨어학과 조교수

자바는 실행속도가 저하라는 심각한 문제점을 갖는다. 이 문제점을 해결하기 위해 번역 방식의 실행 모델이 등장하게 된다. 첫 번째 방법으로 원시 자바 프로그램을 네이티브 코드(목적코드)로 번역하여 실행시키는 방법이다. 그러나 최종 사용자는 원시 프로그램이 아닌 바이트코드를 얻게되므로 대부분의 경우에 적절하지 못하다. 두 번째 방법은 바이트코드를 네이티브코드로 번역하여 실행시키는 오프라인 바이트코드 컴파일 방식이다. 그러나 이는 실행 전에 컴파일이 완료되어야 하므로 동적 클래스 적재(dynamic class loading)를 지원하지 못하는 문제점을 가진다. 마지막으로 JIT는 실행 시간에 필요에 따라 컴파일 하는 방식으로 번역과 최적화에 따른 실행 시간 부담을 갖는다. 그러나 현재는 JIT를 가장 많이 사용하고 있는 추세이다.[1,2] 이와 관련된 기존 연구에는 Jolt, TurboJ, Toba, Harissa, gcj[1,2,3] 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.

자바는 플랫폼 독립적이라는 장점을 갖는 반면 플랫폼 종속적인 작업의 어려움을 가진다. 이를 해결하기 위해 썬 마이크로시스템은 자바 네이티브 인터페이스(JNI)라는 기술을 내놓고 있다. JNI는 프로그래머가 플랫폼 종속적인 기능의 장점을 취할 수 있는 기술이지만 JVM에 대한 전문적 지식이 있는 프로그래머만 사용하도록 요구하고 있다.[4]

본 연구에서는 원시 자바 프로그램 또는 클래스 파일로부터 JNI 기반 네이티브 코드로의 번역하는 방법을 통하여 위에서 서술한 실행속도 문제와 플랫폼 의존적인 작업의 원활한 수행을 이루어 보고자한다. 이를 위해 우선 클래스 파일로부터 JNI 기반 C 프로그램으로 번역하는 번역기를 구현하였으며, 번역된 C 프로그램이 JNI를 프로그램의 기본 틀로 사용할 수 있도록 함으로서 JVM에 대한 전문적 지식을 갖지 않은 프로그래머도 쉽게 JNI 프로그램을 가능하도록 하는 개발 환경을 구축하였다. 더 나아가 기존 시스템이 동적 클래스를 적재할 수 없었던 문제와 Java 2 플랫폼의 지원하지 못하는 문제를 해결하였다.

본 연구를 통하여 JNI의 사용 목적 중 하나인 실행 속도 향상을 위해 자바 네이티브 인터페이스를 사용하는 과정에 사용자가 자바 언어와 C, C++와 같은 컴파일 방식의 다른 언어를 이중으로 사용하여야하는 불편함을 제거하였다. JIT 보다 부분적으로 효율적인 실행을 달성하였으며, 비전문가도 쉽게 JNI 프로그램에 접근할 수 있는 기회를 제공하였다. 마지막으로 C 또는 C++ 등으로 개발된 라이브러리를 자바 환경으로 쉽게 이전할 수 있는 방법이 제공되었다.

본 논문의 2장에서는 본 연구의 기반 기술이라 할 수 있는 혼합형 실행 모델에 대해 알아보고, 3 장에서는 네이티브 코드의 실행 모델 및 자바 프로그램으로부터 네이티브 코드로의 변환 방법에 대해 살펴보며, 4장에서는 본 시스템의 평가 및 사용 사례에 대해 알아본다. 마지막으로 5 장은 결론 및 향후 연구방향에 대해 알아본다.

II. 실행 모델

2.1 네이티브 코드의 실행 모델

자바 프로그램의 최적화된 실행을 이루기 위한 방법에는 여러 가지가 있다. 그의 첫 번째는 원시 자바 프로그램을 목적 코드로 직접 번역하는 방법이다. 그러나 자바 프로그램은 바이트코드 형태로 배포된다. 따라서 일반 사용자는 직접 원시 자바 프로그램을 접하기 어려우므로 특정 플랫폼에 적절한 최적화된 코드를 생성하기 어렵다.

이 문제점은 원시 프로그램 대신 바이트코드를 사용함으로서 해결될 수 있는데 이것이 바이트코드 컴파일러이다. 또한 자바 바이트코드는 거의 원시코드와 유사한 수준의 정보를 가지고 있다. 따라서, 원시코드 컴파일러에 비교할 수 있는 정도의 양질의 코드를 생성할 수 있다는 장점을 가진다. 바이트코드를 목적 코드로 변환하기 위해서는 모든 클래스 파일이 준비되어 있어야 한다. 그러나 자바 언어는 동적 클래스 적재하는 유용한 기능을 가지고 있다. 이는 실행 당시에 반드시 클래스가 필요한 것은 아니며 실제 호출당시에 적재되어 실행되는 기능이다. 오

프라인 바이트코드 컴파일러에서는 동적 클래스 적재 기능을 상실하는 문제점을 가진다.

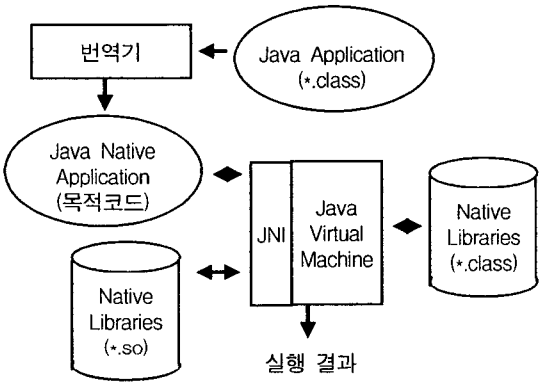
위 문제는 실행 시간에 번역을 함으로서 해결될 수 있는데 이것이 바로 실행 시간 컴파일러 기술인 JIT이다. Just-in-time 컴파일러는 실행 시간에 코드를 번역하며 이는 필요할 때로 한정된다. 컴파일러는 첫 번째 호출된 메소드를 번역하며 이 메소드에서 새로운 클래스의 메소드 호출이 발생한다면 그 때 새로이 적재되어 번역되는 과정을 반복하게 된다. 이러한 방식의 단점은 오프라인 번역 방식과는 달리 실행 시간에 번역을 수행하므로 실행시간 부담으로 발생되며, 더 나아가 시간 소모적인 고급의 최적화 알고리즘을 사용하기 어렵다는 문제점을 가진다.

마지막으로 자바 가상기계를 직접 구현한 자바 프로세서를 고려할 수 있다. 이러한 자바 프로세서는 바이트코드를 목적 코드로 사용하기 때문에 플랫폼 의존적인 언어만큼의 실행 속도를 가질 것이다. 그러나 기존 시스템에 추가적으로 자바 프로세서를 장착해야하는 비용 증가, 하드웨어적으로 구현함으로서 발생하는 유연성의 감소 등 다양한 이유로 인하여 현재 널리 사용되고 있지 않은 실정이다.

본 논문에서 제안한 시스템에서는 바이트코드와 네이티브 이진코드를 혼합하여 실행하는 혼합형 실행 모델에 기반을 두고 있다. 메소드, 필드의 접근, 쓰레드, 가비지 컬렉션 등은 JNI를 통하여 JVM에 의해 이루어진다. 그리고 시간 소모적인 루프, 복잡한 계산, 플랫폼 의존적인 작업 등 네이티브 코드에 의해 수행하는 방식이다. 이 방식은 다음과 같은 장점을 가진다. 사용자가 쉽게 접근할 수 있는 바이트코드를 사용하며, 동적 클래스 적재를 지원하며, JIT에 비해 빠른 실행 시간을 달성할 수 있다.

2.2 혼합형 실행 모델에서 바이트코드의 실행

혼합형 실행 모델은 소프트웨어적 실행에 따른 유연성(flexibility)과 번역 방식에 따른 실행 속도의 향상, 플랫폼 의존적인 작업의 편리한 프로그래밍 방법론 제공이란 장점을 제공한다. 혼합형 실행 모델에서 바이트코드의 실행 형태를 도식화하면 그림 1과 같다.



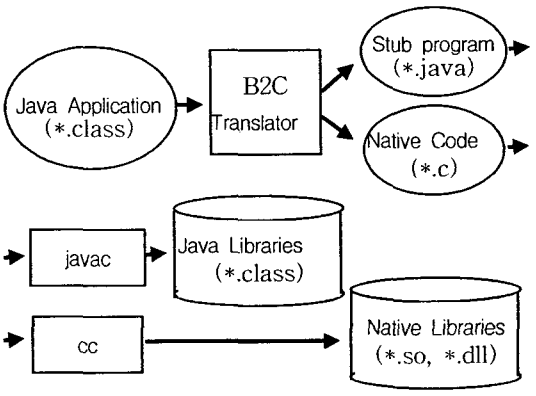
(그림 4) 혼합형 실행 모델에서 바이트코드 실행

그림 1에서와 같이 바이트코드는 번역기 통하여 목적코드로 번역되어진다. 또한 이러한 번역 과정을 거쳐 만들어진 라이브러리는 공유 라이브러리로 구축되어 저장되어질 수 있다. 이는 모두 JNI를 기반으로 구성되며 JNI를 통하여 네이티브 응용프로그램과 자바 가상기계가 대화를 하며 실행된다. 따라서, JVM은 기존의 클래스(*.class)와 네이티브 라이브러리(*.so) 모두를 사용하게 되는 효과를 가지게 된다.

III. 번역기(Translator)의 구성

3.1 번역기의 변환 모델

혼합형 실행 모델과 JNI에 적합한 번역기를 구성하기 위해서 본 연구에서는 JNI를 사용하는 C 코드로 클래스 파일을 번역하고 기존의 C 컴파일러를 사용하여 최적화된 목적코드를 생성하는 방법을 채택하였다. 따라서 번역기는 클래스 파일을 C코드와 이 C 코드를 사용하기 위한 자바 프로그램으로 번역된다. 번역기의 전반적인 변환 절차는 그림 2와 같다.

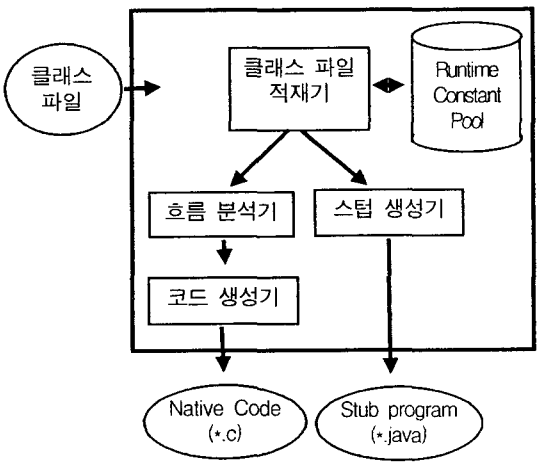


(그림 5) 번역기의 변환 형태

클래스 파일은 B2C Translator에 의해 스텁 프로그램(Stub program)과 네이티브 코드(Native code)로 변환된다. 스텁 프로그램은 자바 응용 프로그램을 시작하기 위한 자바 프로그램이며, 네이티브 코드는 자바 응용 프로그램의 각 메소드를 C의 함수 형태로 갖고 있는 C 프로그램이다. 이렇게 변환된 스텁 프로그램은 javac에 의해 JVM에서 실행 가능한 일반 클래스 파일로 변환되며, 네이티브 코드는 기존의 C 컴파일러에 의해 JVM와 JNI를 통하여 대화할 수 있는 공유라이브러리 형태인 *.so 또는 *.dll 형태로 변환된다.

3.2 B2C Translator의 구성

B2C Translator는 클래스 파일을 스텁 프로그램과 네이티브 코드로 변환하여 주는 역할을 한다. B2C Translator의 구성 형태는 그림 3과 같다. 이를 위해서는 클래스 파일 적재기는 클래스 파일을 읽어 실행 시간 상수 풀(runtime constant pool)을 구성한다. 이렇게 읽어들인 정보를 바탕으로 스텝 프로그램과 네이티브 코드를 생성하게 된다. 스텁 생성기는 상수 풀 정보를 이용하여 스텁 프로그램을 생성하게 되는데 스텁 프로그램은 자바 응용프로그램의 필드와 메소드의 시그너춰는 유지하면서 메소드의 구현은 네이티브 코드로 대치한 자바 프로그램이다.



(그림 6) B2C Translator의 구성

흐름 분석기는 바이트코드에 대한 제어의 흐름을 분석하여 레이블, 예외처리, 지역변수의 결정을 위한 정보를 수집하게 된다. 흐름 분석기

는 상수 풀로부터 바이트코드를 순차적으로 읽어들이면서 정보를 수집하게 된다. 조건부 점프에 해당하는 ifeq, ifne, iflt, ... 등, 그리고 무조건 점프에 해당하는 goto, goto_w, 예외처리를 위한 jsr, ret, 다중 점프를 위한 tableswitch, lookupswitch에 대해 점프의 시작위치와 목적지 그리고 점프의 개수 등 흐름에 대한 정보를 수집한다. 이러한 정보는 코드 생성 시에 코드 생성을 위한 레이블, 예외처리, 임시 변수의 생성 등을 위해 사용하게 된다.

수집된 흐름 정보를 바탕으로 하여 코드 생성기는 바이트코드를 C 코드로 변환하게 된다. 자바 가상기계의 명령어는 스택에 기반을 두고 있다. 따라서 스택에 있는 원소의 개수는 필요한 임시 변수의 개수와 동일하다. 따라서 스택 top 정보를 이용한다면 불필요한 임시 변수의 생성을 막을 수 있다.

식 a1 = a1 + b;에 변환 형태를 보면 그림 4와 같다.

바이트코드	생성된 C 코드

iload_1
iconst_5
iadd
istore_1
	stack_0.i=local_1.i;
	stack_1.i=5;
	stack_0.i=stack_0.i+stack_1.i;
	local_1.i = stack_0.i;

(그림 9) 식 a1 = a1 + 5;의 변환

그림 4의 생성된 C 코드에서 모든 임시 변수는 stack_n.i의 형태를 가지며 여기서 n은 현재 스택의 top에 해당하는 숫자이며 뒤의 .i는 type을 의미하며 실제 저장될 변수이다.

조건문 if (tmp == 0) flag = false;의 변환 형태를 알아보자.

바이트코드	생성된 C 코드

iload_3
ifne 32
iconst_0
istore 4
xxx
	stack_0.i = local_3.i;
	if (stack_0.i != 0) goto L32;
	stack_0.i = 0;
	local_4.i = stack_0.i;
	L32: .....

(그림 5) if (tmp == 0) flag = false;의 변환

자바의 필드 접근은 JNI 메소드를 통하여 이루어진다. 클래스 변수의 접근을 위해서는 GetStatic<Type>Field(), 인스턴스 변수를 위해서는 Get<Type>Field()를 통하여 접근하게 된다. 이와 유사한 방법으로 메소드의 호출을 위해서는 CallStatic<Type>Method()와 Call<Type>Method()를 사용하여 자바 메소드를 호출하게 된다. 예외처리 및 동적 배열을 처리하기 위해서는 Throw()와 New<Type>Array(), NewObjectArray() 등을 사용한다. 서술한 변환 방법에 따라 메소드 호출 System.out.println("Main program ends");의 변환결과는 다음과 같다.

```
tmpclazz=(*env)->FindClass(env, "java/lang/
                System");
tmpfid=(*env)->GetStaticFieldID(env, tmpclazz,
                "out", "Ljava/io/PrintStream;");
stack_0.l=(*env)->GetStaticObjectField(env,
                tmpclazz, tmpfid);
stack_1.l=(*env)->NewStringUTF(env,
                "Main program ends");
tmpclazz=(*env)->FindClass(env, "java/io/Print
                Stream");
tmpmid=(*env)->GetMethodID(env, tmpclazz,
                "println", "(Ljava/lang/String;)
                V"); (*env)->CallVoidMethod
```

(env, stack_0.l, tmpmid, stack _1.l);

클래스 파일을 이용하여 네이티브 코드와 스텁 프로그램을 생성하기 때문에 원시 프로그램을 요구하지 않는다. 따라서, 최종 사용자가 쉽게 얻을 수 있는 바이트코드로부터 플랫폼 의존적인 환경에서의 실행 장점을 얻을 수 있다.

3.3 스텁 프로그램과 네이티브 코드의 생성

스텁 생성기에 의해 생성된 스텁 프로그램의 기본 형태는 그림 6과 같다.

```
/* Stub java program */
public class Java_App extends
        java.lang.Object {
  // 필드 리스트
        /* Java_App의 field list를 그대로
           가짐 */
        ....
  // 네이티브 메소드 리스트
  native public void _B2C_init(...);
        /* Java_App의 field list를 그대로
           가짐 */
    ....
  // 생성자
  public Java_App(...) {
        _B2C_init(...);
  }
  // 정적 초기화
  static {
    System.loadLibrary("Java_App");
        _B2C_clinit();        // Option
  }
}
```

(그림 6) 스텁 프로그램의 기본 구조

생성된 스텁 프로그램은 필드 리스트, 네이티브 메소드 리스트, 생성자, 그리고 정적 초기화(static initializer)로 구성된다. 필드 리스트는 변환 전, 원시 자바프로그램의 필드 리스트와 동일한 형태를 가진다. 네이티브 메소드 리스트는 클래스 메소드 또는 인스턴스 메소드이며, 네이티브 메소드로 변환된다. 생성자는 네이티브 메소드일 수 없으므로 이는 스텁 프로그램에 의해 재 생성되며 기존의 생성자는 일반 네이티브 메소드로서 처리된다. 마지막으로 정적 초기화에서는 공유라이브러리의 적재, 클래스 변수에 대한 초기화, 원시 자바프로그램의 정적 초기화의 처리 등이 이루어진다. 정적 초기화는 원시 자바프로그램의 특성에 따라 생성되므로 나타나지 않을 수 있다. .

완전수를 구하는 프로그램과 그의 스텁 프로그램 그리고 생성된 코드의 일부는 그림 7, 8, 9와 같다.

```
public class Perfect {
  public void doit(int arg) {
  int max = arg, i, j, k, r, sum, i = 2;
   while ( i <= max) {
     sum = 0; k = i / 2; j = 1;
     while ( j <= k) {
        r = i % j;
        if (r == 0) sum += j;
        j += 1;
     } // end while
     if ( i == sum)
          System.out.print(i + ", ");
     i += 1;
    } // end while
  }
  public static void main(String args[]) {
    Perfect p1 = new Perfect();
    int val = Integer.parseInt(args[0]);
    p1.doit(val);
    System.out.println("");
  }
}
```

(그림 7) Perfect 수를 구하는 프로그램

```
/* Stub java program */
public class Perfect extends java.lang.Object
{
   native public void _B2C_init() ;
   native public void doit(int arg1) ;
   native public static void
    main(java.lang.String[] arg1) ;
   public Perfect(){ _B2C_init(); }
   static {System.loadLibrary("Perfect");}
}
```

(그림 8) 그림 7에 대한 스텁 프로그램

```
/* C program */
#include <jni.h>

#include "b2c_runtime.h"
union {unsigned int bytes; float fval;}
            _b2c_float_inf={0x7F800000};

JNIEXPORT void JNICALL
Java_Perfect__1B2C_1init(JNIEnv* env, jobject
            obj) {
jvalue stack_0;
....
jobject exceptionRefs;
local_0.l = obj;
/*   0 : aload_0 */
   stack_0.l = local_0.l;
...

JNIEXPORT void JNICALL
Java_Perfect_doit(JNIEnv* env, jobject obj,
            jint arg1) {
jvalue stack_0, stack_1, stack_2, stack_3;
....
jfieldID tmpfid;
jobject exceptionRefs;
local_0.l = obj;
local_1.i = arg1;
/*   0 : iload_1 */
   stack_0.i = local_1.i;
/*   1 : istore_2 */
   local_2.i = stack_0.i;
/*   2 : iconst_2 */
...
```

(그림 9) 그림 7에 대한 네이티브 코드

IV. 평가 및 사용사례

실험 환경은 Intel Pentium III 500, II 300의 CPU와 RAM 256, 128 환경의 하드웨어, 운영체제는 RedHat Linux 6.0과 Win32 Windows를 사용하였고, 자바는 JDK 1.3.0, C 컴파일러는 gcc와 MS Visual Studio 6.0을 이용하였다. JIT 컴파일러는 실행시간에 컴파일과 최적화에 대한 부담을 갖기 때문에 본 시스템에 의해 생성된 프로그램보다 실행속도가 떨어진다. 실험 예제로 단순 loop 반복, Perfect number, Permutation 등의 예제를 통하여 이를 확인하였으며 그의 실험결과는 표 1과 같다. 그러나 순환함수(Permutation)의 경우는 JVM과의 상호작용이 증가하는 관계로 실행속도 저하를 가져왔다.

<표 1> 기존환경과 혼합형 실행 모델의 성능 평가

Benchmark program	기존환경	혼합모델	실험환경
Perfect Number (10000)	8.95 sec 2.39 sec 3.90 sec	2.15 sec 2.38 sec 3.73 sec	Pentium III 500 Linux, nojit Pentium III 500 Linux, jit Pentium II 300 win32, jit
Nested loop (400)	5.24 sec 1.51 sec 2.58 sec	1.24 sec 1.47 sec 1.10 sec	Pentium III 500 Linux, nojit Pentium III 500 Linux, jit Pentium II 300 win32, jit
Permutation java (recursion test, 7)	0.35 sec 0.57 sec 0.93 sec	0.57 sec 1.05 sec 1.43 sec	Pentium III 500 Linux, nojit Pentium III 500 Linux, jit Pentium II 300 win32, jit

바이트코드를 스텁 자바프로그램과 JNI를 이용한 네이티브 코드로 번역하고, 혼합형 실행기법을 사용하여 동적으로 바이트코드를 적재할

수 있는 능력을 유지하면서, JIT 컴파일러 보다 더 효율적인 실행속도를 가진다.

생성되는 C 프로그램은 자바네이티브 인터페이스를 사용함으로써 이를 지원하는 모든 자바 가상 기계에서 실행될 수 있으며, 네이티브 메소드를 작성하는 경우 헤더 파일 생성 도구의 사용과 매개변수로 전달되는 객체에 대한 처리 부담을 줄일 수 있는 부가적인 효과도 가진다. 다음의 플랫폼에 의존적인 작업을 자바로 작성하는 경우를 고려하여 보자. 그러나 자바는 하드웨어 제어와 같은 플랫폼 의존적인 작업을 수행하지 못하므로 이는 네이티브 코드로 작성해야 한다. 이 때 그림 10과 같이 자바 프로그램을 작성하여 변환하면 그림 11과 같은 C 프로그램이 생성되므로 네이티브 코드 프로그래머는 JNI에 관한 전문적 지식 없이 단지 "// C 코드 ☜" 부분에 C로 프로그래밍 하여 사용할 수 있으므로 네이티브 메소드 개발 환경으로서 사용 가능하다.

```
public class HwControl {
  int state;
  public static int p_control(int port) {
    int retval=0;
      // C 코드 ☜
    return retval;
  }
  ....
  public static void main(String[] arg) {
    int retval;
    retval=p_control(23);
  }
  ....
}
```

(그림 10) 네이티브 코드로 작성할 자바프로그램

```
....
NIEXPORT jint JNICALL
Java_HwControl_p_1control
 (JNIEnv* env, jclass cls, jint arg1) {
jvalue stack_0;
jvalue local_0, local_1;
jclass tmpclazz;
jobject tmpobj;
jmethodID tmpmid;
jfieldID tmpfid;
jobject exceptionRefs;
local_0.i = arg1;
    stack_0.i = 0;
    local_1.i = stack_0.i;
    // C 코드 ☜
    stack_0.i = local_1.i;
return stack_0.i;
}
```

(그림 11) 생성된 C 프로그램

V. 결론

본 논문에서는 원시 자바 프로그램 또는 클래스 파일로부터 JNI 기반 네이티브 코드로의 번역하는 방법을 통하여 실행속도 문제와 플랫폼 의존적인 작업의 원활한 수행을 이루어 보고자 하였다. 이를 위해 우선 클래스 파일로부터 JNI 기반 C 프로그램으로의 번역기를 구현하였으며, 번역된 C 프로그램을 JNI 프로그램을 위한 기본 틀로 사용할 수 있도록 함으로서 JVM에 대한 전문적 지식을 갖지 않은 프로그래머도 쉽게 JNI 프로그램을 가능하도록 하는 개발 환경을 구축하였다. 더 나아가 기존의 동적 클래스를 적재할 수 없는 문제와, Java 2 플랫폼의 지원하지 못하는 문제를 해결하였다.

본 연구를 통하여 JNI의 사용 목적 중하나인 실행 속도 향상을 위해 자바 네이티브 인터페이스를 사용하는 과정에 사용자가 자바 언어와 C, C++와 같은 컴파일 방식의 다른 언어를 이중으

로 사용하여야하는 불편함을 제거하였다. 또한 JIT 보다 부분적으로 효율적인 실행을 달성하였으며, 비전문가도 쉽게 JNI 프로그램에 접근할 수 있는 기회를 제공하였다. 마지막으로 C 또는 C++ 등으로 개발된 라이브러리를 자바 환경으로 쉽게 이전할 수 있는 방법이 제공되었다.

현재는 혼합형 실행 모델에서 네이티브 메소드 개발을 위해 사용자는 명령어 라인 형태로 작업해야 한다. 앞으로의 향후 과제로 혼합형 실행 모델을 위한 네이티브 프로그램 개발 환경을 GUI 형태로 구축함으로서 사용자가 보다 쉽게 사용토록 하고자 한다.

VI. 참고문헌

[1] Todd A.Proebsting et al., 'Toba : Java For Applications - A Way Ahead of Time (WAT) Compiler' Proc.Conf. bject-Oriented Technologies and Systems (COOTS '97), Usenix Assoc., Berkeley, Calif., 1997.

[2] Gills Muller and Ulrik Pagh Schultz 'Harissa : A Hybird Approach to Java Execution', IEEE Software, V.16 N.2, 44-51, 1999.

[3] GCJ Document http://gcc.gnu.org/java/docs.html

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

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

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

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

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

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

Mixed-mode execution environment using the JNI

Sang-Hoon, Kim*

Abstract

The tradeoff of Java's portability is the inefficiency of interpretation. Also, the standard Java class library may not support the platform-dependent features needed by your application. Several solutions have been proposed to overcome these problems, such as JNI, JIT, off-line bytecode compilers. In this paper, we present an mixed-mode execution model which Java virtual machine executes together with native code. This execution model and its translator preserves the ability to dynamically load bytecode, and reduce the difficulties of JNI usages. Our system is more efficient than JIT, and helps programmer to write C implementation for the native method without the concept of JNI.

* Dept. of software Semyung Univ.