정보학연구 제3권 제2호 2000년 6월(pp.13-21)
The Studies of Information Technology, Vol.3, No.2

자바 네이티브 메소드 생성 시스템

김도영*/김상훈**

요 약

자바 네이티브 메소드는 시간 소모적인 작업의 효율적 실행, 플랫폼 종속적인 작업의 수행, 라이브러리의 재사용 등을 위해 제안되었다. 실행 속도 향상을 위해 네이티브 메소드를 사용한다면, 네이티브 메소드 구현에 자바 언어가 아닌 컴파일 방식의 다른 언어를 사용해야한다. 또한 자바 네이티브 인터페이스의 개념을 습득해야한다. 이러한 부담을 경감시키기 위해 본 논문에서는 자바 메소드를 네이티브 메소드로 자동 변환하여 주는 자바 네이티브 메소드 생성 시스템을 제안하였다. 또한 본 시스템은 JNI의 개념을 요구하지 않기 때문에 C 언어로 네이티브 메소드를 작성하려는 프로그래머에게 편의를 제공한다.

Ⅰ. 서론

자바는 플랫폼에 독립적으로 실행될 수 있도록 설계된 객체-지향 언어이다. 이는 한 번 컴파일 된 코드는 다시 재 컴파일 과정과 수정 없이 어떠한 플랫폼에서도 실행가능하고, 하드웨어 또는 운영체제에 독립적인 언어라는 것을 의미한다. 자바의 플랫폼 독립성은 자바의 클래스를 클래스 파일(class file)이라고 불리는 형태로 컴파일하는 것으로부터 시작된다. 클래스 파일은 자바 가상 기계(Java virtual Machine : JVM)내의 명령어들 형태인 바이트 코드(bytecode)와, 자바 클래스에 관한 정보를 포함한다. 번역된 클래스 파일은 자바 가상 기계를 지원하는 모든 컴퓨터에서 실행 가능하다. 그러나 자바 언어가 기계 독립적인 특징인 높은 이식성을 가지는 반면에 바이트 코드를 소프트웨어적으로 구성된 JVM에서 처리되기 때문에 느린 실행속도를 갖는 단점이 있다.

자바의 이러한 단점을 보완하기 위해서 JIT (Just-in-time)와 오프라인 바이트코드(offline bytecode) 컴파일러[1,2]와 같은 몇 가지 방법들이 제안되었다. JIT 방식은 메소드를 호출 직전에 해당 클래스 파일을 불러와 이를 네이티브 코드로 번역한 후 수행한다. 여기서, 사용되지 않는 코드는 컴파일하지 않으므로 불필요한 번역시간의 부담을 줄일 수 있고, 컴파일 시간과 실행 시간의 차이를 제거할 수 있으나, 프로그램 실행동안 컴파일을 수행하게 되므로 프로그램 실행 시간에 부담이 될 정도의 최적화(optimization)는 수행하지 못한다. 오프라인 컴파일러는 최적화를 필요한 만큼 충분히 수행할 수 있으며, 실행시간 부담을 갖지 않으므로 더 효율적인 실행이 가능하다. 그러나 컴파일 시간에 모든 바이트 코드를 요구하므로 동적으로 바이트 코드를 적재할 수 있는 능력에 제한을 가지게 된다. 위의 두 종류의 해결 방법은 서로간의 장단점을 가진다.

실행속도의 향상이란 측면은 자바 네이티브

* 세명대학교 대학원 전산정보학과
** 세명대학교 소프트웨어학과 조교수

인터페이스(Java Native Interface : JNI)를 통하여서도 가능하다[4]. 본 논문에서는 다양한 플랫폼에서 실행 가능한 자바의 장점과 비교적 빠른 실행 속도를 가지는 기존 컴파일 방식 언어의 장점을 갖는 자바 네이티브 메소드 생성 기(Native Method Generator : NMG)를 구현한다. 이는 실행 속도 향상을 위해 자바 네이티브 인터페이스를 사용하는 과정에 사용자가 자바 언어와 C, C++와 같은 컴파일 방식의 다른 언어를 이중으로 사용하여야하는 불편함을 제거하면서 JIT 보다 더욱 효율적인 실행을 달성하는데 그 목적이 있다. 자바 메소드를 네이티브 메소드로의 변환은 번역된 바이트 코드로부터 C 언어로 구성된 네이티브 메소드를 자동 생성하도록 구현되었다. 생성된 네이티브 코드는 C의 함수이므로 내부에 프로그램을 자유로이 삽입할 수 있다. 따라서, 실행 속도의 향상뿐만 아니라 플랫폼 종속적인 작업의 수행을 위해 자바 네이티브 메소드 생성 시스템을 사용한다면 사용자는 자바 네이티브 인터페이스에 대해서는 인식할 필요가 없다는 장점을 가진다.

2장에서는 본 연구의 기반 기술이라 할 수 있는 JVM에서 네이티브 메소드 호출, JNI, 바이트 코드의 C 변환 등에 대해 알아본다. 3 장에서는 본 연구에서 제안한 NMG의 구조 및 수행과정에 대해 살펴보며, 마지막으로 4 장에서는 본 시스템의 성과 및 보안점 그리고 향후 연구방향에 대해 알아본다.

II. 관련 연구

2.1 JVM에서 네이티브 메소드의 호출

자바 가상 기계는 자바 클래스 파일을 실행하는 스택 기반의 가상 기계로 정의한다.[5] 각 실행상태의 쓰레드는 메소드를 호출 시 push, 복귀시 pop되는 새로운 엑티베이션 레코드(activation record)인 자신만의 자바 스택을 가진다. 이 자바 스택은 프로그램의 수행 도중 어떤 메소드가 호출되었는가를 기록하고, 각 메소드의 호출과 관련된 데이터들의 상태 정보를 포함하고, 지역변수와 피연산자 스택, 그리고 프로그램 카운터를 포함하고 있다. 모든 계산은 피연산자 스택에서 수행되고, 임시 결과들은 지역 변수 내에 저장된다. 자바 메소드의 호출 방식은 다음과 같다. 활성 매개변수(actual parameters)들은 호출이 이루어지기 전에 호출자 메소드의 피연산자 스택에 push된다. 가상 메소드 호출의 경우 명시적인 this 참조자는 또한 첫 번째 매개변수로서 push된다. 자바 가상 기계는 이런 매개변수들을 pop하고 피호출 메소드의 지역변수 배열로 그들을 이동시킨다. 자바 메소드가 복귀될 때, 피호출자의 피연산자 스택의 top에 위치하는 복귀 값은 호출자의 피연산자 스택의 top으로 이동된다.

자바 가상 기계 내에는 자바 스택과 네이티브 스택간에는 호출 관계가 형성되고, 본 논문에서 제시하고 있는 네이티브 메소드 생성기에서 생성된 C 코드와 클래스 파일들, 그리고 공유라이브러리는 필요시 자바 가상 기계를 통해서 서로간의 호출이 이루어진다. 자바 메소드와 네이티브 메소드의 호출관계는 (그림 1)과 같다. 네

이티브 메소드 스택은 다른 자바 가상 머신 메소드와 같은 방식으로 사용된다. 그러나 이 스택을 사용하는 메소드들은 자바 가상 머신의 명령어들이 아닌 다른 언어를 사용해서 구현된다. 자바 가상 기계는 네이티브 메소드를 지원하기 위해서는 C 스택이라 불리는 스택을 사용한다. 이는 C와 같은 언어에서 자바 가상 머신의 명령어들을 에뮬레이션(emulation)하는데 쓰인다.

(그림 1)에서와 같이 자바 스택과 네이티브 메소드 스택 사이에는 호출이 일어난다. 자바 메소드는 네이티브 메소드를 호출할 수 있고, 네이티브 메소드는 자바 메소드를 호출할 수 있다.

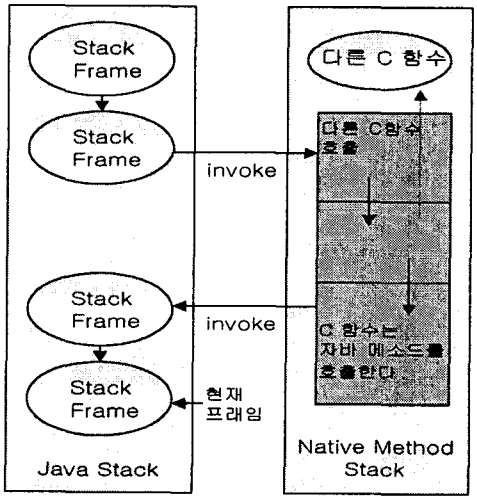

그림 1. 자바 스택과 네이티브 메소드 스택간의 관계

자바 메소드가 네이티브 메소드를 호출할 때 자바 메소드는 자바 가상 기계의 상태의 정보를 전달한다. 그리고 네이티브 메소드가 자바 메소드를 호출 할 때 네이티브 스택은 이전의 네이티브 메소드의 반환을 요구한다.

2.2 네이티브 메소드의 호출

프로그램이나 라이브러리를 작성하는데 있어서 자바가 모든 것을 해주지는 못한다. 많은 양의 기존 코드가 이미 사용되고 있을 경우에 자바로 처음부터 다시 작성하는 것보다 기존의 코드들을 연결하여 사용하는 것이 더 효율적이고, 실행속도가 중요한 프로그램에 자바 환경의 속도가 충분하기 않을 경우에, 다른 언어로 구현하는 것이 보다 효율적일 것이다. 이를 위하여 자바는 NMI를 제공하며 사용자는 이를 이용하여 자바 네이티브 메소드를 이용할 수 있다. (그림 2)는 JDK 1.0.x에서 C를 이용한 네이티브 메소드의 실행 과정을 나타내고 있다. 자바 프로그램을 위한 네이티브 메소드를 생성하고 실행하는 경우에는 다단계에 걸친 과정이 필요하다.

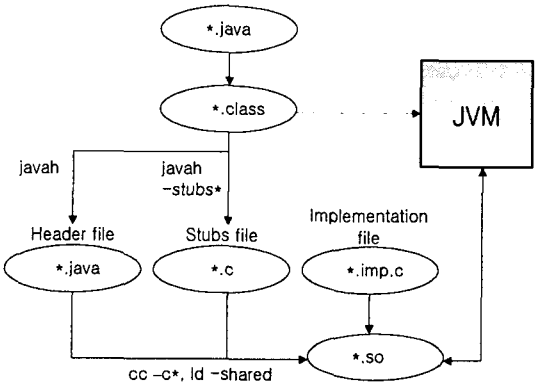

그림 2. JDK1.0.x에서의 자바 NMI 실행 과정

자바 프로그램을 작성해서 네이티브 메소드를 선언하는 자바 클래스를 작성하고, 자바 클래스를 컴파일해서 헤더 파일을 생성한 후 네이티브 메소드의 구현부분 C코드를 작성한다. 마지막으로 공유라이브러리 파일로 헤더파일과 구현된

파일들을 컴파일하고 실행한다.

위와 같은 과정으로 네이티브 메소드를 컴파일하고 실행한다. 네이티브 메소드를 이용한 자바 프로그램의 수행은 본래의 자바 프로그램에 비하여 실행 속도는 증가하나 자바의 다양한 플랫폼을 넘나드는 속성을 잃게 될 것이다.

2.3 바이트코드를 C로의 변환

자바 NMI는 자바 클래스 파일의 실행 속도를 향상시키기 위하여 시스템에 의존적인 원시 코드를 필요로 한다. 이러한 원시 코드 프로그램은 자바에 비하여 빠른 실행 속도를 가지는 프로그래밍 언어를 이용하여 사용자가 직접 작성하여야 하지만, 본 논문에서 제안하는 자바 NMG는 자바로 작성된 프로그램을 직접 원시 코드로 변환하는 오프라인 자바 - 네이티브 컴파일러를 포함한다.

이처럼 자바 클래스 파일의 바이트 코드를 원시 코드로 변환하는 오프라인 자바-네이티브 컴파일러(Java-to-Native Compiler)의 설계에 있어서 두 가지 사항을 고려할 수 있다. 즉, 해당 플랫폼만의 런-타임 서비스를 제공하면서 모든 것을 실행할 수 있는 독립적인 코드를 생성할 것인가 아니면 특정 서비스만을 위하여 JVM과 연결될 원시 코드를 생성할 것인가 하는 문제이다. 그리고 오프라인 자바-네이티브 컴파일러는 형태에 있어서 두 가지로 분류할 수 있다. 즉, 네이티브 컴파일러와 논-네이티브(non-native) 컴파일러이다. 네이티브 컴파일러는 직접 실행 가능한 코드를 생성하는 반면에, 논-네이티브 컴파일러는 중간 언어 형태의 코드를 생성한다. 네이티브 컴파일러의 설계의 이점은 두 가지가 있다. 첫째, 생성된 이진 코드는 중간 언어 형태의 코드보다 더 효율적이며, 둘째, 연속적인 툴(tool) 들이 요구되지 않기 때문에, 컴파일 속도가 빠르다. 이러한 기법의 단점으로는 이식이 가능하지 않고, 효율적인 코드의 생성은 목적 프로세서에 대한 확장된 정보를 요구한다는 것이다. 논-네이티브 컴파일러는 이보다 유연하고 경쟁적인 성능을 제공한다. 이러한 바이트 코드를 C 코드로 변환하는 것에 대한 기존의 연구로는 Toba, Turboj, Jolt, Harrissa등이 연구되었다.[1,2]

III. Native Method Generator

NMI를 이용하는 것은 자바의 실행 속도를 향상시킬 수 있으나, 플랫폼에 상관없이 실행 가능한 자바 클래스 파일의 특성이 손실되는 단점을 안고 있다. 또한, 사용자가 NMI를 이용함에 있어 실제적인 프로그램의 처리는 C 언어와 자바 언어를 모두 습득해야 한다는 불편함을 가지고 있다.

이러한 단점을 보완하기 위하여, 자바 네이티브 메소드 자동화 도구인 자바 네이티브 메소드 생성기 NMG를 설계하고 구현했다. 즉, NMI를 이용하면서, 사용자가 두 가지 종류의 언어를 다루어야 하는 불편함을 해소하고, NMI의 존재를 의식할 필요 없는 편리한 사용을 위해서, 자바 네이티브 메소드를 자동으로 생성해 주는 자동화 도구 NMG를 제안한다.

본 논문에서 제안된 자동화 도구에서 바이트 코드와 목적 코드(target code)의 중간 코드로 C 언어를 사용하고자 하는 이유는 개발자는 프로세서와 서브프로세서 사이에 존재하는 잠재적인 차이점을 언급할 필요가 없고, 대부분의 플랫폼에 C컴파일러가 존재하기 때문이다. 또한 기존

에 연구된 컴파일 기법과 최적화 기법의 이용이 용이하며, 또한 최적화에 따른 자바 가상 기계의 부담을 경감시킬 수 있다.

NMG는 일반 자바 프로그램을 번역하는 부분과 네이티브 메소드로 변환되기를 원하는 자바 프로그램을 처리하여 C로 작성된 네이티브 메소드로 변환하고 목적코드로 컴파일하는 부분으로 구성된다.

(그림 3)은 NMG의 구조 및 실행 모습을 보여주고 있다. main method 부분과 called method는 NMG의 입력인 두 자바 프로그램이다. main method는 javac로 컴파일되어 네이티브 메소드를 이용하기 위한 class 파일을 생성하게 되고, called method는 C 코드로 번역될 자바 프로그램이다. called method는 Stub Class Generator에 의해서 네이티브 메소드를 호출하기 위한 *.class가 생성되고, 실제 자바로 구성된 실행 코드는 번역기에 의해서 *.c가 번역된다. 이 *.c는 called method에서 생성된 *.class를 입력으로 받아서 번역기에 의해 생성된 C 프로그램이다. 이렇게 생성된 *.c 프로그램을 이용하여 자바 네이티브 메소드에서 이용 가능한 공유 라이브러리인 *.so를 생성한다.

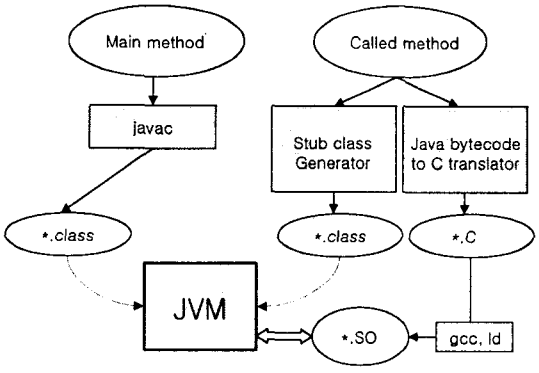

그림 3. NMG의 구조 및 실행

main method를 컴파일 한 *.class와 Stub class Generator에 의해 생성된 *.class와 *.so는 JVM에 의해 사용되어 프로그램이 수행되어진다.

```
<<main method>>
package abc;
public class LoopTime{
  public static void main(String argv[]){
        int count = 0;
        int res;
        System.loadLibrary("tnative");
        LoopBody c = new LoopBody();
        count = Integer.parseInt(argv[0]);
        res = c.doIT(count);
        System.out.println("count => " +
count + " Result => " + res);
    }
}

<<called method>>
package abc;
public class LoopBody {
        public int doIT(int cnt) {
        int sum = 0;
              for(int i=0; i < cnt; i++)
                 for(int j=0; j < cnt; j++)
                   for(int k=0; k < cnt;
k++)
                       sum = sum + 1;
        return sum;
        }
}

<<Stub code>>
package abc;
```

```
public class LoopBody{
    public native int doIT(int i);
    public LoopBody() { }
}

<<translator에 의해 변환된 C 코드의 일부>>
stack_item   *Java_abc_LoopBody_doIT_stub
(stack_item *_P_,struct execenv *_EE_){
cp_item_type *_cp;
struct fieldblock *_fb;
struct methodblock *_mb;
int32_t var_0 = _P_[0].i;
int32_t var_1 = _P_[1].i;
int32_t var_2;
int32_t var_3;
.....
if (SelfRef == ((void *)0) )
 { SelfRef = FindClass(_EE_, "abc/Loop-
Body", TRUE); }
_cp = SelfRef->constantpool;
stack_1 = 0;
var_2 = stack_1;
stack_1 = 0;
var_3 = stack_1;
goto label_44;
label_7: stack_1 = 0;
var_4 = stack_1;
goto label_35;
label_13: stack_1 = 0;
var_5 = stack_1;
goto label_26;
label_19: stack_1 = var_2;
stack_2 = 1;
stack_1 += stack_2;
var 2 = stack_1;
var_5 += 1;
label_26: stack_1 = var_5;
stack_2 = var_1;
if (stack_1 < stack_2) {goto label_19;}
var_4 += 1;
label_35: stack_1 = var_4;
stack_2 = var_1;
if (stack_1 < stack_2) {goto label_13;}
var_3 += 1;
label_44: stack_1 = var_3;
stack_2 = var_1;
if (stack_1 < stack_2) {goto label_7;}
stack_1 = var_2;
_P_[0].i = stack_1;
return (_P_ + 1);
method_exit: return _P_;
}
```

그림 4. 예제 프로그램

(그림 4)는 NMG에 대한 예제 프로그램을 나타내고 있다. 프로그램 main method는 네이티브 메소드를 호출하는 프로그램으로 javac로 컴파일 되어서 *.class를 생성하고, called method는 네이티브 메소드로 변환되어질 부분으로 Stub class generator에 의해 *.class가 생성되고, 번역기에 의해 네이티브 코드로 변환되어진다.

Ⅳ. 결론 및 향후 연구 과제

자바의 장점인 다양한 플랫폼에서 실행가능하다는 것은 바이트코드를 이용하기 때문에 가능하며, 바이트코드를 실행하기 위해서는 인터프

리터 방식을 이용하고 있다. 인터프리터 방식은 실행속도 저하를 가져오게 되며, 호스트 의존적인 작업을 수행하거나 기존 라이브러리의 사용 등이 자바의 문제점으로 나타나고 있다. 느린 실행 속도를 개선하기 위하여 JVM은 JIT를 사용하고 있으나 아직은 부족한 실정이다. 또한 시간 소모적인 루틴의 실행속도 향상, 기존에 제작된 라이브러리의 재사용, 호스트 의존적인 작업의 수행 등의 작업을 위해 JNI의 지원을 받아 해결하고 있으나 어느 정도의 불편이 따른다.

본 논문에서는 자바의 메소드를 네이티브 메소드로 변환하는 것이 자동적으로 이루어지도록 함으로써, 시간 소모적인 루틴의 실행 속도 향상을 위해 컴파일 방식 언어인 C나 C++를 사용해야 하는 개발자의 불편을 완화하며, 자바 네이티브 메소드 인터페이스의 존재를 인식하지 않고 쉽게 자바 네이티브 메소드를 사용할 수 있도록 하기 위해 NMG를 제안했다. NMG는 다양한 플랫폼에서 실행 가능한 자바의 장점과 비교적 빠른 실행 속도를 가지는 기존의 컴파일 방식 언어의 장점을 동시에 갖추고 있다. 실행 속도 면에서도 JDK 1.2의 JIT보다 NMG를 사용하여 실행시키면 LINUX 환경의 경우 1.5 ~ 8배까지의 실행 속도 향상을 보았다. 이는 컴파일과 실행을 동시에 수행하게되는 JIT의 단점으로 인한 당연한 결과이다. NMG에서의 바이트코드를 C 코드로 번역하는 번역기는 기존의 Harissa, TurboJ, Jolt, J2C등 기존의 개념을 이용하였고, JDK 1.0.2의 환경에서 구성하였으며, 현재 JDK 1.0.2 이상의 환경에서 실행 가능하도록 하는 연구가 계속 진행되어지고 있다.

앞으로의 향후 연구 과제는 코드 최적화를 C 컴파일러에 전적으로 의존하기보다는 자바 바이트코드의 C 코드로의 변환 과정에서 코드 최적화를 실행하는 연구가 이루어져야 할 것이고, JIT 컴파일러와 네이티브 메소드를 병행하여 사용함으로써 실행속도를 더욱 향상시킬 수 있는 연구가 필요하다. 그리고 이 논문에서는 고려하지 않았지만, 바이트코드의 최적화를 고려 시 자바 가상 기계를 좀 더 효율적으로 이용하는 방법에 대한 연구가 향후 계속적으로 진행되어야 할 것이다.

V. 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-+, 1999.

[3] Gills Muller et al., 'Harissa : A Fexible and Efficient Java Environment Mixing Bytecode and Compiled code', http://www.irisa.fr/compose/hairssa/harissa.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] Joshua Engel, 'Programming for the Java Virtual Machine', Addison Wesley, 1999

[7] Mike Cohn, Bryan Morgan et al, 'Java Developer's Reference', Sam's net, 1997

[8] Toba homepage, URL:http://wwwsunos4.nada.kth.se/javadoc/toba

[9] TurboJ Documentation, URL:http://hpk.felk.cvnt.cz/turboj/contents.html

[10] Ken Arnold, James Gosling, 'The Java Programming Language, Second Edition., Addison Wesley., 1998.

Java Native Method Generating System

Do-Young, Kim*/Sang-Hoon, Kim**

Abstract

Java native method is proposed for the efficient execution of time-critical code, running of platform dependent job, and reuse of established libraries. If the writing of the Java native method is the speedup of execution time, you must use a compiled language not java language to write native method. Also, you must know the usage of the Java native interface to use native method. To reduce these difficulties, we proposed java native method generator that changes java method into native method automatically. Also, NMG helps programmer to write C implementation for the native method because there is no need for the concept of JNI.

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