Journal of Korea Multimedia Society Vol. 13, No. 10, October 2010(pp. 1514-1524)

자바와 C/C++의 혼합 프로그래밍

김 상 훈†

요 약

표준 자바 클래스 라이브러리는 응용 프로그램에 의해 요구되는 플랫폼 의존적인 기능을 수행하지 못한다. 따라서 플랫폼 의존적인 기능을 포함한 자바 응용 프로그램은 JNI를 사용한 네이티브 함수를 호출하여 부족한 기능을 수행하여야 한다. 네이티브 언어 프로그래머는 다양한 자바 객체와 연결하고 사용 후 이를 해지해야하는 작업을 명시적으로 기술해야 한다. 이러한 번거로운 작업을 피하기 위한 방법을 본 논문에서 제안하고자 한다. 순수 자바 클래스에서 네이티브 메소드는 코드 블록을 가질 수 없다. 자바 네이티브 메소드가 네이티브 코드로 작성된 몸체를 가질 수 있도록 함으로써 프로그래머는 JNI를 의식하지 않고 프로그래밍하는 것이 가능하다. 이를 위해 네이티브 환경의 자바 클래스인 네이티브 클래스라는 개념, 그리고 자바 클래스와 네이티브 클래스 간에 제어와 자료의 교환을 지원하는 중재기를 두어 해결하였다.

Java and C/C++ Mixed Programming

Sang-Hoon Kim†

ABSTRACT

The standard Java class library does not support the platform-dependent features needed by the application. Therefore, the Java application including the platform -dependent features must supplement the required features by invoking native functions using JNI. The native language programmer has to explicitly specify how to connect to various Java objects and later to disconnect from them. In this paper, I suggest a way to avoid these annoying works. The native method in the pure java class can not contain a native code block. By providing a native code block for the native method, it is possible for programmer to write a native code without being aware of JNI. To achieve this, I introduced the native class that is a java class on the native environment, and made it possible to interchange data by placing an arbitrator between the java class and the native class.

Key words: Java(자바), C/C++, Mixed programming(혼합 프로그래밍), JNI

1. 서 론

자바 언어는 플랫폼 독립적이란 특성으로 인하여 다양한 분야에서 널리 사용되고 있다[1]. 그러나 표준 자바 클래스 라이브러리는 응용 프로그램에서 요구하는 플랫폼 의존적인 기능을 지원하지 못한다. 또한 기존에 네이티브 언어로 작성된 라이브러리의 재사용에 어려움을 가진다. 마지막으로 실행시간이 매우 중요한 요소인 응용분야에 부적합하다. 이러한 문제점을 해결하기 위해 썬 마이크로시스템즈는 자바 네이티브 인터페이스(Java Native Interface: JNI)라는 기술을 제시하였다[2,3]. 그러나 JNI 기술을 사용하기 위해서는 자바 가상 기계(Java Virtual Machine: JVM)의 내부 구조와 JNI에 대한 지식을 요구한다[4]. 또한 JVM과 정보 교환을 위해서는 다수의 전처리와 후처리 과정이 필요하다. 따라서 JNI에 대

※ 교신저자(Corresponding Author) : 김상훈, 주소 : 충북 제천시 세명로 117번지 세명대학교 컴퓨터학부(390-711), 전화 : 043)649-1266, FAX : 043)649-1700, E-mail : kimsh@semyung.ac.kr

접수일 : 2010년 7월 20일, 수정일 : 2010년 9월 6일
완료일 : 2010년 9월 17일

† 정회원, 세명대학교 컴퓨터학부 부교수

한 충분한 지식을 가지고 있더라도 각종 JNI 함수의 적절한 선택과 사용은 시간 소모적이고 지루한 작업이다. 이는 소프트웨어 개발 생산성과 품질 저하의 요인이 된다.

가상 기계를 통한 간접 실행으로 야기되는 실행 속도 저하라는 문제점은 하드웨어의 발전, JIT (Just-In-Time) 번역 기술 그리고 최적화 기술 등의 발전으로 많은 성능 향상을 이루었다. 기존 라이브러리의 재사용이 어렵다는 문제점은 GlueGen, Hawt-JNI, JNA 등의 네이티브 라이브러리 재사용 도구가 다수 개발되어 있어 개발자에게 도움을 주고 있다 [5-7]. 그러나 표준 자바 환경에서 플랫폼 의존적인 작업은 여전히 수행할 수 없다. 따라서 플랫폼 의존적인 작업은 JNI를 사용하는 네이티브 코드로 작성하여야 한다. 관련 연구로 C++ 템플릿 클래스를 이용하여 네이티브 코드 작성의 어려움을 해결하려한 사례가 있다[8]. 또한 라이브러리 재사용 도구는 네이티브 함수 호출 및 매개변수에 대한 전처리 및 후처리 코드를 자동으로 생성하여 주기 때문에 네이티브 코드의 작성에 도움을 받을 수 있다. 그러나 새로운 패키지 또는 라이브러리의 사용해야하는 부담을 가지게 되며, 라이브러리 재사용 도구의 경우에는 자바 클래스의 필드와 메소드의 접근에 제약을 가진다.

자바 클래스는 코드 블록을 갖지 않는 순수 네이티브 메소드 선언만을 포함할 수 있다. 본 연구에서는 자바 클래스에 네이티브 코드를 직접 작성할 수 있도록 하여 네이티브 코드 작성의 어려움을 해소하고자 한다. 또한 자바 어노테이션(annotation)을 이용하여 네이티브 언어의 종류와 네이티브 코드에서 필요한 헤더 파일 정보를 전달하고자 한다. 본 연구의 결과인 네이티브 코드 블록을 가지는 자바 클래스를 JaC(Java class with C/C++ block)라 부르도록 하겠다.

JaC 클래스를 설계함에 있어 다음 사항을 중점적으로 고려하였다. 우선 기존 자바 언어와 C/C++ 개발자가 추가적 학습 부담을 최소화하여야 한다. 둘째 JaC 클래스가 기존 JNI를 사용한 자바 클래스에 비해 추가적인 실행시간 부담이 없어야 한다. 이를 달성하기위해 JaC 클래스는 네이티브 메소드 선언이 C/C++ 코드 블록을 가질 수 있다는 것을 제외하면 순수 자바 클래스와 동일하다. 따라서 자바 자료형과 C/C++ 자료형의 차이점만 인식한 상태에서 JaC 클래스를 작성할 수 있다. 필드와 메소드 ID를 반복하여 검색하는 것을 방지하기 위해 ID 값을 저장하여 재사용하는 기법을 사용하였다. 그리고 자바 객체와 네이티브 객체 간의 상태 동기화를 네이티브 함수 종료 시점에 일괄 처리하여 실행 시간 부담을 최소화하였다.

2장은 네이티브 메소드 작성의 어려움과 이를 해결하기 위한 관련 연구를 통해 본 연구의 필요성을 보여준다. 네이티브 코드 블록을 가지는 자바 클래스인 JaC 클래스의 문장구조와 자료형 변환에 대해 3장에서 알아본다. 4장에서는 JaC 클래스의 실행 방법과 번역기에 대해 기술한다. 5장 평가 및 결론에서는 JaC 클래스의 작성 사례, 그리고 성능 평가를 통해 본 연구의 타당성 및 연구 성과에 대해 알아본다.

2. 관련 연구

자바 언어가 가지는 플랫폼 의존적인 작업과 기존 라이브러리의 재사용에 어려움을 해결하기 위한 기술이 JNI이다. 그러나 JNI를 사용하기 위해서는 다양한 전처리 및 후처리 과정을 요구하여 프로그래머를 괴롭히고 있다. 본 장에서는 이러한 어려움은 무엇이며, 이를 해결하기 위한 다양한 지원 도구들에 대해 알아보고, 그들의 한계점에 대해 살펴봄으로써 본 연구의 필요성을 보이고자 한다.

JNI를 사용하여 자바 네이티브 메소드를 구현하는 과정은 다음과 같다. 네이티브 메소드를 구현한 함수는 JNI 명세에 따라 변형된 함수 원형을 사용하여야 한다. 그리고 네이티브 메소드의 매개변수로 전달된 참조형 자바 객체는 네이티브 코드에서 접근할 수 있는 형태로 변환한 후 접근하여야 한다. 또한 필드와 메소드의 접근은 멤버 ID를 얻어온 후, 이를 통해 자바 객체를 얻어오고, 다시 네이티브 코드에서 접근할 수 있는 형태로 변환하는 과정을 거쳐야 한다. 매개변수나 필드의 사용을 종료한 후에는 이 사실을 JVM에 통보하여 변경된 자료의 반영 및 사용 자원을 반납하는 후처리 과정을 수행해야 한다. 그림 1은 네이티브 메소드에서 필드 값을 출력하고 필드의 값을 변경하는 프로그램 예제로 JNI 명세서 4.1절에 나와 있다[1].

그림 1의 네이티브 메소드는 단순히 필드 s를 접근하여 그 내용을 출력하고 필드 s의 값을 "123"으로

```
IFAcc.java
public class IFAcc {
    private String s;
    private native void accF();
    public static void main(String args[]) {
        IFAcc c = new IFAcc();
        c.s = "abc";
        c.accF();
        System.out.println("Java: c.s = \"" + c.s + "\"");
    }
    static { System.loadLibrary("IFAcc"); }
}
IFAcc.c
JNIEXPORT void JNICALL
Java_JFAcc_accF(JNIEnv *env, jobject obj) {
    jfieldID fid; jstring jstr; const char *str;
    jclass cls = (*env)->GetObjectClass(env, obj);//①
    fid = (*env)->GetFieldID(env, cls, "s",
                        "Ljava/lang/String;");//②
    jsr=(*env)->GetObjectField(env, obj, fid);//③
    str=(*env)->GetStringUTFChars(env, jstr, NULL);//④
    printf("C: c.s = \"%s\"\n", s);//⑤
    (*env)->ReleaseStringUTFChars(env, jstr, NULL);//⑥
    jsr = (*env)->NewStringUTF(env, "123");//⑦
    (*env)->SetObjectField(env, obj, fid, jstr);//⑧
}
```

그림 1. 필드 접근 네이티브 메소드

변경하는 프로그램이다. 이를 위해 객체 클래스 얻기, 필드 ID 얻기, 필드 값 얻기, C 언어 환경에서 접근 가능한 형태로 자료 변환하기, 필드 내용 출력하기, 필드 사용 해제하기, 새로운 스트링 객체 생성하기, 생성된 객체를 필드 값으로 설정하기 등 여러 단계의 과정을 거쳐야 한다. 이는 단순하고 반복적인 작업이며 개발 생산성 및 소프트웨어 품질을 떨어트리는 주요 요인이 된다.

네이티브 메소드 구현의 어려움을 해결하기위해 C++의 템플릿 기능을 사용하여 캡슐화를 시도한 연구 "JNI - C++ integration made easy"가 있다[8]. 이 연구에서는 C++의 constructor와 destructor가 각각 전처리와 후처리를 담당하고 있다. 그림 1의 IFacc.c를 이 연구에서 제시한 방법으로 다시 프로그래밍하여 보면 다음과 같다. 문장 ①, ②, ③, ④, ⑥은 JNIStringUTFChyars s(env, obj, "s");로 대치될 수 있고, 문장 ⑤는 cout << s.get();, 문장 ⑦, ⑧은 JNIField<jstring>(env, obj, "s") = env->New-StringUTF("123");으로 대치된다. 그러나 이 방법은 변형된 함수 헤더를 사용하는 문제와 위 템플릿 클래스의 사용법을 익혀야 하는 추가 부담, 마지막으로 C++ 템플릿을 기반으로 하고 있어 네이티브 언어로 C 언어를 사용할 수 없다는 한계점을 가진다.

GlueGen은 자바 코드로부터 C 라이브러리를 호출을 위해 필요한 자바 코드와 JNI 코드를 자동으로 생성하여 주는 도구이다[5]. 이는 C 헤더 파일과 GlueGen 설정 파일을 읽어 들여 C 코드와 자바 클래스를 출력한다. 이를 사용하여 매개변수로 전달된 자바 객체를 C 코드에서 접근할 수 있는 형태로 변환해야 하는 부담을 줄일 수 있어 쉽게 네이티브 코드와 연결이 가능하다. 예를 들어 float process_data (float* data, int n);을 포함한 function.h 헤더파일과 패키지, 클래스 이름, 파일 위치 정보를 포함한 function.cfg를 입력으로 받은 GlueGen이 출력한 자바 코드와 JNI 코드는 그림 2와 같다.

```
TestFunction.java
TestFunction.java
package testfunction;
public class TestFunction {
static public native float process_data(float[] data, int n);
}
TestFunction_JNI.c
TestFunction_JNI.c
JNIEXPORT jfloat JNICALL
Java_testfunction_TestFunction_process_1data(
JNIEnv *env, jclass _unused, jfloatArray data, jint n) {
float * _ptr0 = NULL;
float _res;
if (data != NULL) {
_ptr0 =
   (float *) (*env)->GetPrimitiveArrayCritical(env, data, NULL);
}
_res = process_data((float *) _ptr0, (int) n);
if (data != NULL) {
(*env)->ReleasePrimitiveArrayCritical(env,data,_ptr0,JNI_ABORT);
}
return _res;
}
```

그림 2. GlueGen이 자동 생성한 프로그램

HawtJNI는 네이티브 메소드를 구현하기에 필요한 JNI 코드를 생성하여 주는 코드 생성기이다[6]. JNI 코드에 관한 어노테이션 정보와 hawtjni 패키지를 사용한 자바 클래스를 입력으로 받아 JNI 코드를 생성하여준다. GlueGen과 마찬가지로 네이티브 함수를 호출하고 자바 객체를 접근하고 전달하는데 필요한 전처리 및 후처리 작업을 수행하는 JNI 코드를 자동으로 생성하여 준다.

JNA(Java Native Access)는 자바 프로그램이 JNI의 도움 없이 네이티브 공유 라이브러리를 쉽게 접근할 수 있도록 개발된 라이브러리이다[7]. JNA는 일반 자바 메소드 호출 방법을 사용하여 네이티브

함수를 직접 호출할 수 있다. 이를 위해 개발자는 네이티브 라이브러리 내에 있는 함수와 자료구조를 기술하는 자바 인터페이스를 사용해야 한다.

전통적인 방법으로 네이티브 메소드를 구현하는데 다수의 전처리와 후처리과정을 요구한다. C++의 템플릿 라이브러리는 자바 필드와 메소드의 접근이 자유롭다. 그러나 클래스의 사용법을 습득해야 하는 추가적 부담과 네이티브 코드로 C 언어를 사용할 수 없다는 문제점을 가진다. GlueGen, HawtJNI, JNA과 같은 지원 도구들은 자바 패키지의 사용 또는 외부 파일을 통하여 네이티브 클래스에 대한 정보를 제공해야하는 부담을 가진다. 또한 네이티브 환경에서 자바 필드와 메소드의 접근에 대한 고려가 없거나 또는 빈약하다는 단점을 가진다.

본 연구에서는 이상에서 언급한 연구 및 기존 도구들에서 발생하는 부담을 해결하고자 자바 클래스 내에 네이티브 코드를 직접 작성할 수 있는 방법을 제안하였다. 본 논문에서 제안한 방법은 네이티브 코드로 C와 C++를 모두 사용할 수 있고, C++ 템플릿 라이브러리 또는 자바 패키지의 사용이라는 추가적인 부담을 갖지 않으며, 자바의 필드와 메소드를 모두 자유로이 접근할 수 있다.

3. JaC 클래스

JaC 클래스는 자바 클래스와 C/C++ 함수라는 두 개의 언어 환경이 존재하게 된다. 이 두 언어를 결합하기 위해서는 새로운 구문의 도입과 자료형의 연결이 필요하다. 본 장에서는 JaC 클래스의 문장구조와 자료형의 매핑에 대해 알아본다.

JaC 클래스는 네이티브 메소드가 네이티브 코드 블록을 가지는 자바 클래스이다. 자바 메소드가 네이티브 메소드를 호출하면 제어가 자바 환경에서 네이티브 환경으로 이동하게 된다. 이 때 네이티브 환경에서 자바 환경의 모든 속성들을 자유로이 접근하기 위해서는 모든 자바 속성들을 네이티브 환경에서 참조 가능하여야 한다. 이를 위해 본 연구에서는 네이티브 클래스 개념을 도입한다. 네이티브 클래스는 네이티브 환경의 자바 클래스이다. 논문을 서술함에 있어 이 두 환경에서 사용하는 용어의 혼동을 피하기 위해서 표 1과 같이 용어를 정의하고자 한다.

자바 환경의 용어는 자바 언어에서 일반적으로 사용하는 용어이며, 네이티브 환경의 용어는 본 논문을 명료하게 설명이 가능하도록 정의하였다. 스텁 함수(stub function)와 프락시 함수(proxy function)는 두 환경 상호간에 제어 및 자료의 전달을 도와주는 역할을 담당하는 중요 함수로 4.1절에서 자세히 다루고 있다.

표 1. 용어 정의

자바 환경	네이티브 환경
자바 클래스	네이티브 클래스
자바 필드	네이티브 필드
자바 메소드	네이티브 자바 함수
네이티브 메소드	네이티브 함수
스텁 함수(CDT) : 네이티브 함수 호출자	
프락시 함수(CDT) : 자바 메소드 호출자	

우선 JaC 클래스의 전제적인 문장구조와 제어의 흐름에 대해 그림 3을 통해 살펴보자. JaC 클래스의 C 코드 블록 작성자는 필드 및 매개변수의 자료형이 언어 관점에 따라 다르다는 것, 네이티브 메소드의 구현 부분(그림 3의 점 선 사각형)을 C/C++ 언어로 작성한다는 것, 그리고 선택적으로 어노테이션(annotation)을 사용하여 헤더 파일과 사용 언어를 지정할 수 있다는 것을 제외하면 자바 클래스를 작성하는 것과 동일하다. 즉, 네이티브 메소드가 네이티브 코드로 작성된 몸체를 가진다는 사항을 제외하면 자바 문법과 동일하다. 어노테이션은 속성으로 include와 lang을 가지며 그의 선언 형태는 다음으로 가정된다.

```
public @interface NativeCode {
        String include() default "";
        String lang() default "C";
}
```

속성 include는 네이티브 메소드에서 사용하는 각종 선언을 포함하는 헤더 파일을 기술하는 부분으로 여러 개의 헤더파일이 필요한 경우 세미콜론(;)에 의해 분리 표현될 수 있다. 속성 lang은 네이티브 언어의 종류를 기술하기 위한 부분으로 기술하지 않으면 묵시적으로 C이며, 명시적으로 C 또는 C++의 사용이 가능하다. 물론 lang 속성으로 C++를 사용하면 선언이 문장 내 임의 지점에 나타날 수 있으며, 함수 중복(function overloading)이 가능하고, 엄격한 형

검사(type checking) 수행이 가능해지는 등 다양한 C++ 언어의 장점을 취할 수 있다. 그림 3의 실선 화살표 위에 있는 단일 원 숫자 ①, ②, ③은 프로그래머 입장의 개념적 접근 경로이며 이는 일반 자바 프로그램과 동일하다. 점선 화살표 위의 이중 원 숫자 ①, ②, ③은 단일 원 숫자 ①, ②, ③에 대응하는 내부 접근 경로이며 각각의 접근은 네이티브 메소드 중재기(Native Method Arbitrator: NMA)를 경유하여 간접적으로 이루어진다. NMA는 두 개의 다른 언어 간의 접근을 투명하게 도와주는 중재기로 4.1절에서 상세하게 다룬다.

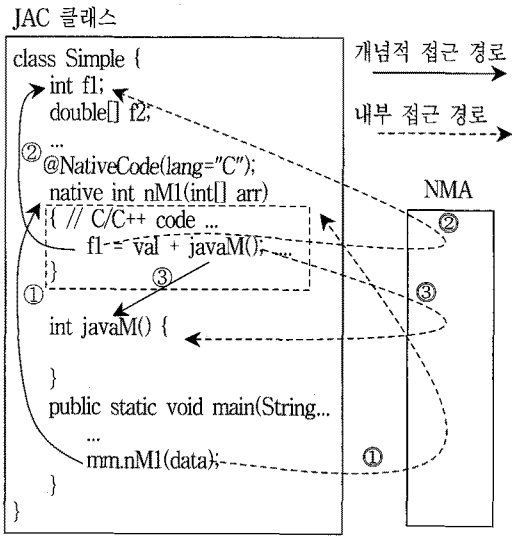

그림 3. JAC 클래스 구조와 멤버의 접근

자바 프로그램을 이루는 기본 구성단위는 클래스이다. 자바 언어는 C 언어에 존재하지 않는 클래스, 필드, 메소드 등을 가지고 있으므로 이에 적절한 개념 매핑이 필요하다. 클래스는 관련된 멤버를 모아 캡슐화를 이루는 단위이다. 이에 대응하는 C 언어의 개념은 파일 영역을 통해 달성 가능하다. 이들 개념의 매핑 관계를 표 2이 보여주고 있다. C 언어에서 외부 선언 수정자로 static을 사용하면 이름 영역은 파일 영역으로 한정된다. 물론 C++의 class 또는 namespace를 사용하면 더욱 훌륭한 매핑 관계를 이룰 수 있다. 그러나 이러한 개념의 도입은 네이티브 코드로 C 언어를 사용할 수 없게 된다. 따라서 본 연구에서는 C와 C++를 모두 사용할 수 있도록 파일과 static을 사용하였다.

표 2. 자바와 C/C++의 구문 매핑

자바 환경	네이티브 환경
클래스	파일
필 드	외부 정적 변수(파일영역)
메소드	외부 정적 함수(파일영역)

C 언어와 자바 언어 사이에 자료형의 대응관계에 대해 살펴보자. 자바 언어의 자료형은 기본 자료형과 참조형으로 나누어진다. 자바의 기본 자료형은 C 언어와 구조적으로 유사하며 그의 매핑 관계는 표 3과 같다. 자바와 C 언어에서 기본 자료형의 값은 모두 자료 값(data value)이다. 곧 변수나 상수의 값이 모두 해당 자료를 나타내는 비트 표현인 것이다. 또한 두 언어 모두 매개변수 전달 방식이 값 전달 방식(call by value)으로 매개변수가 해당 자료형의 값을 포함하고 있다.

표 3. 기본 자료형과 String 매핑

자바	C	자바	C
boolean	unsigned char	long	long long
byte	signed char	float	float
char	unsigned short	double	double
short	short	void	void
int	long(or int)	String	const char*

자바 언어의 참조형에는 배열, 클래스, 인터페이스가 있다. 모든 자료형을 C 언어로 대응시키기에는 너무 복잡해지고 시스템에 부담을 초래할 가능성이 많다. 따라서 JaC 클래스에서는 네이티브 메서드의 사용 목적을 손상하지 않는 범위 내에서 배열과 스트링 클래스로 사용 범위를 제한한다. 네이티브 메소드로 스트링의 전달이나 스트링 자료형 필드의 접근은 표 3에서와 같이 C 언어의 const char *로 단순 대응관계를 가진다. 그러나 자바의 배열은 C 언어 배열과 달리 원소의 개수를 가지는 length 속성을 가진다. length 속성을 지원하기 위해 자바 배열은 C 언어의 구조체에 대응시켰다. 자바 실수형 배열 double[] f2는 C 언어 관점에서는 다음과 같은 형태를 가지는 것으로 간주한다.

```
typedef struct {
    double *value;
```

```
    int length;
} DoubleArray;
DoubleArray f2;
```

C 언어에서 배열의 이름은 실제로 배열의 시작 주소 상수이다. 따라서 배열을 double 형 포인터 변수로 가정하면 의미적으로 동일하다. 실제 사용 예를 살펴보기 위해 배열 필드 f2의 내용을 모두 더하여 반환하는 프로그램 그림 4를 살펴보자.

```
Simple.java
class Simple {
double[] f2;
...
  native double sumArray() {
      int idx;
      double sum=0;
      for(idx = 0; idx < f2.length; idx++)
          sum += f2.value[idx];
      return sum;
  }
```

그림 4. 배열 필드의 접근

그림 4와 같이 배열 필드 f2를 value 와 length 속성을 가진 구조체 변수로 고려하여 프로그래밍하면 된다. 실수형 배열 자료를 sumArray(double[] f2)와 같이 매개변수로 전달 받았다할지라도 동일한 메소드 몸체를 가진다. 나머지 자료형에 대해서도 구조체 멤버 value 자료형이 표 4에 따라 변경하는 것을 제외하면 double 형 배열과 동일한 방법으로 사용 가능하다.

지금까지 네이티브 메소드 몸체에서 자료형에 따른 매개변수 또는 필드의 접근에 대해 알아보았다.

표 4. value 속성 자료형과 C 구조체

자바	value 속성	C 구조체
boolean[]	unsigned char *	BooleanArray
byte[]	char *	ByteArray
char[]	unsigned short *	CharArray
short[]	short *	ShortArray
int[]	long *(or int *)	IntArray
long[]	long long *	LongArray
float[]	float *	FloatArray
double[]	double *	DoubleArray

다음은 네이티브 메소드 몸체에서 C 자료를 자바 객체로 반환하거나, 또는 자바 메소드를 호출하고 반환값을 얻는 경우에 대해 알아보자. 이 모든 경우 해당 자료의 자료형이 기본형이거나 스트링인 경우는 표 3의 자료형 매핑 관계에 따라 자료를 주고받을 수 있다. 그러나 네이티브 메소드 반환 자료형이 배열인 경우, 자바 메소드 호출에서 배열 형 매개변수를 사용하는 경우, 그리고 자바 메소드 호출의 결과로 반환되는 배열 자료형인 경우는 주의해야 한다. 자바 객체로 배열을 전달하거나 반환 값으로 전달 받기 위해서는 표 4에 따라 대응하는 자료형으로 해당 배열을 생성한 후 자료를 주고받아야 한다. 그림 5는 자바 메소드 setter를 통해 배열 필드에 값을 배정하는 예이다.

배열 필드를 접근하여 배열의 원소 값을 참조하고 변경하는 것은 가능하다. 또한 배열이 초기화되어 있지 않거나 또는 새로운 배열 객체를 배정하는 경우는 예제에서 보여주는 것과 같이 setter setData(int[] data)를 사용하거나, setter 대신 data = val;을 사용하여 배정 가능하다. 자바 객체의 필드 값이 자바 객체에 실제로 반영되는 시점은 네이티브 메소드의 종료 시점이다. 필드 값 변경을 실시간으로 반영하기 위해서는 시스템에 너무 많은 부담을 주므로 메소드 종료 시점에 일괄적으로 처리하고 있다. 따라서 네이티브 필드의 값이 변경되었다고 그에 대응하는 자바 필드의 값이 즉시 변경되는 것이 아니고 네이티브 메소드의 종료 시점에 변경된다는 것에 주의해야 한다.

```
TestNative.java
class TestNative {
    int[] data;
    ...
    void setData(int[] data) { this.data = data; }
    native int sendArray() {
        int values[] = {1, 3, 5, 7, 9};
        IntArray val = {values, 5};
        setData(val);
        ...
    }
```

그림 5. 배열 필드에 값 배정

4. Jacc 시스템 구성

이전 장에서 JaC 클래스의 문장구조와 자료형 그리고 자료의 접근에 대해 살펴보았다. 본 장에서는 자바 객체와 네이티브 메소드의 상호작용을 지원하

는 NMA와 이를 생성하는 JaC 클래스 번역기에 대해 살펴본다.

4.1 네이티브 메소드 중재기(NMA)

NMA는 자바 클래스에 대응하는 네이티브 클래스와 제어 및 자료 전달기(Control and Data Transferrer: CDT)로 이루어진 파일 단위의 네이티브 환경이다. 네이티브 클래스는 네이티브 필드, 네이티브 자바 메소드 그리고 네이티브 함수로 이루어진 C 언어 클래스이다. 네이티브 필드와 네이티브 자바 메소드는 자바 클래스의 필드와 메소드에 대응하는 네이티브 환경의 필드와 메소드이다. 그리고 네이티브 함수는 자바 클래스에 존재하는 네이티브 메소드에 대응하는 네이티브 환경의 함수이다. CDT는 프락시 함수, 스텁 함수 그리고 각종 부가 루틴들로 이루어진 제어 및 자료 전달기이다. 그림 3의 Simple JaC 클래스를 위한 NMA의 구조는 그림 6과 같다.

자바 객체와 네이티브 함수의 상호 작용은 세 가지로 나누어 살펴볼 수 있다. 하나는 자바 메소드에서 네이티브 함수의 호출이고, 둘째는 네이티브 함수에서 필드의 접근이며, 마지막은 네이티브 함수에서 자바 메소드의 호출이다.

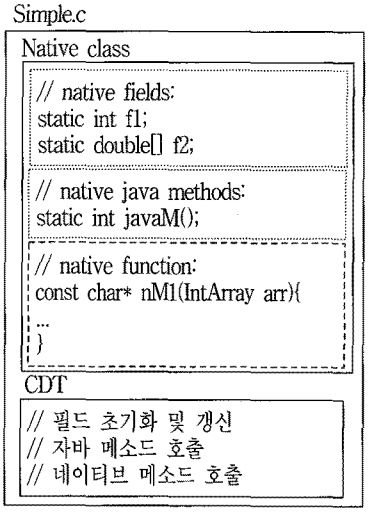

그림 6. NMA의 구조

자바 메소드가 네이티브 함수를 직접 호출할 수 없다. 이는 스텁 함수를 경유하여 간접 호출이 이루어진다. 이 스텁 함수는 실제로 JNI 명세에 따른 네이티브 메소드 구현이다. 네이티브 함수에서 필드를 접근하기 위해서는 그의 참조가 네이티브 언어 환경 내에 미리 정의되고 초기화되어 있어야 한다. 이는 자바 메소드가 네이티브 메소드를 호출하는 시점에 자바 필드가 네이티브 환경으로 이동되어 네이티브 필드를 초기화되어야 한다. 이러한 작업은 스텁 함수에 의해 이루어진다. 네이티브 함수에서 자바 메소드의 호출을 살펴보자. 네이티브 함수가 직접 자바 메소드를 호출할 수 없다. 이는 프락시 함수가 대리로 자바 메소드를 호출하여 준다. 실제로 이 프락시 함수는 네이티브 자바 메소드이다. 이러한 상호 작용은 자료의 생성 또는 변경을 동반하게 된다. 네이티브 함수의 종료 시점에 네이티브 환경에서 생성된 자료가 자바 객체로 전달되며 변경사항이 자바 객체에 반영된다. 변경된 자바 객체의 실재 반영 시점이 네이티브 메소드 종료 시점으로 연기되기 때문에 동일 객체를 다양한 경로를 통해 접근하는 경우 최후에 변경한 사항만 반영된다. 이러한 문제점을 해결하기 위해 일괄처리 대신 실시간 변경을 한다면 시스템의 부담이 너무 커지는 문제점을 가진다. 또한 동일 객체의 다양한 경로 접근은 일반 프로그래밍에서 사용되지 않는 방법이므로 JaC 클래스에서는 일괄처리 방식을 택하고 있다.

NMA를 구성하는 CDT의 핵심 모듈인 스텁 함수와 프락시 함수에 대해 좀 더 구체적으로 살펴보자. 자바 메소드가 네이티브 함수를 호출하는 경우 매개변수로 전달되는 자료와 반환으로 돌려받는 값에 대해 고려하여 보자. 두 언어의 매개변수 전달 방식은 모두 값 전달 방식을 사용하고 있으므로 표 3와 같이 대응하는 자료형으로의 단순 변환만으로 직접 접근 가능하다. 자바 참조형은 C의 입장에서 보면 모두 포인터이다. 참조형 자료는 직접 접근은 불가능하며 JNI을 경유하여 C 언어에서 접근할 수 있는 형태로 변환해 주어야 한다. 본 연구에서는 자바 참조형 중에서 스트링과 기본형 배열만 접근이 가능하도록 하고 있다. 매개변수로 전달된 자바 스트링은 JNI 함수인 GetStringUTFChars을 사용하여 C 스트링으로 변환하여 읽어 온다. 이때 자바 스트링의 자료형은 jstring이며 C 스트링의 자료형은 const char* 이다. 반환된 스트링의 사용을 종료하면 ReleaseString-UTFChars를 호출하여 C 스트링을 위해 사용된 메모리를 free해 주어야 한다. 스트링과 마찬가지로 배

열도 C 환경에서 직접 접근하여 사용할 수 없다. C 환경에서 기본형 배열을 직접 접근할 수 있도록 배열로의 포인터를 얻어오기 위해 JNI 함수 Get<Type>ArrayElements를 사용하고, 사용을 마친 후 이 사실을 JVM에 알리고 변경사항을 반영하기 위해 Release<Type>ArrayEmements를 호출해야 한다. C 언어의 배열과는 달리 자바 배열은 원소의 개수를 나타내는 length 속성을 가진다. 이 차이점을 극복하기 위해 매개변수로 전달된 배열은 배열로의 포인터를 가지는 value 속성과 길이를 나타내는 length 속성을 가지는 구조체로 변환된다. 각 자료형 배열에 따른 구조체는 표 4에서 보여 주었다. 다음으로 네이티브 함수 실행 결과로 반환 받은 값의 자료형이 기본형인 경우는 두 언어의 자료형 구조가 동일하므로 매개변수와 마찬가지로 동일하게 반환할 수 있다. 그러나 C 언어의 스트링과 배열은 New<Type>Array와 NewStringUTF를 사용하여 자바 언어의 스트링과 배열로 생성하여 반환하여야 한다. 그림 7은 네이티브 메소드 선언, 스텁 함수, 네이티브 함수의 관계와 매개변수 전달 및 자바 객체 반환에 어떻게 이루어지는지를 보여주고 있다.

네이티브 메소드 선언 - 자바 클래스

```
native int nM1(int[] arr);
```

스텁 함수 - CDT

```
JNIEXPORT jint JNICALL Java_Simple_nM1
(JNIEnv* env, jobject obj, jintArray j_arr) {
    // ① 매개변수, 필드 준비
    cRetVal = nM1(c_arr);
    // ② 변경된 자바 객체의 반영 및 반환 객체 생성
    return jRetVal;
}
```

네이티브 함수 - 네이티브 클래스

```
static int nM1(IntArr arr) { ... }
```

그림 7. 스텁 함수 - 네이티브 함수 호출자

네이티브 함수에서 자바 객체의 필드를 접근하는 방법에 대해 알아보자. 자바 필드는 매개변수와는 달리 기본 자료형도 직접 접근이 불가능하다. 필드의 값을 참조하기 위해서는 GetFieldID와 Get<Type>Field를 사용하여 필드의 값을 네이티브 환경으로 가져온 후 네이티브 필드를 초기화해야 한다. 또한 네이티브 함수에서 네이티브 필드의 값을 변경하였다면 Set<Type>Field를 사용하여 변경 사항을 자바 필드에 반영시켜야 한다. 스트링, 배열과 같은 참조형은 GetFieldID와 GetObjectField를 사용하여 자바 jstring와 j<Type>Array형의 자바 객체를 가져온다. 이렇게 얻어진 자바 객체를 네이티브 자료형으로 변환하여 네이티브 필드를 초기화 한다. 변환 방법은 매개변수 변환 방법과 동일하다. 그림 7-①에서는 네이티브 필드 f1, f2를 초기화하며 네이티브 함수의 매개변수 c_arr를 준비하는 작업을 수행한다. 그림 7-②에서는 f1, f2, c_arr의 변경 사항을 자바 객체 j_arr에 반영하며 반환받은 공간을 해제하는 작업을 행한다. 또한 네이티브 함수의 반환 값을 자바 객체로 변환하여 반환해야 한다. 그러나 현재의 예는 기본형인 int로 구조가 동일하여 cRetVal을 jRetVal로 직접 배정하여 반환하면 된다. 네이티브 함수의 반환 자료형이 참조형이라면 cRetVal에 해당하는 자바 객체를 생성한 후 이 값을 jRetVal에 배정하여 반환하는 과정을 가진다.

마지막으로 네이티브 함수에서 자바 메소드를 호출하는 경우에 대해 알아보도록 하자. 첫째 GetObjectClass와 GetMethodID를 호출하여 메소드 식별자를 얻어야 한다. 그 다음 반환 자료형이 Type인 메소드를 호출하기 위해서는 Call<Type>Method를 사용하여 호출하게 된다. 이와 함께 네이티브 함수가 자바 메소드와 주고받는 자료의 변환을 함께 처리해 주어야 한다. 이 모든 과정은 프락시 함수가 담당하여 처리하고 있다. 실제로는 프락시 함수의 순수 선언이 네이티브 자바 메소드이다. 네이티브 자바 메소드의 정의 부분인 프락시 함수는 매개변수로 전달된 자료의 변환, 자바 메소드 호출, 반환된 자료의 변환이란 3가지 작업을 수행하게 된다. 첫 단계인 그림 8-①은 프락시 함수가 전달 받은 매개변수를 자바 메소드가 접근할 수 있는 자바 객체로 변환하는 과정이다. 기본 자료형은 표 2의 자료형 매핑 테이블에 따라 단순 형 변환으로 직접 전달 가능하다. 그러나 C 언어의 스트링과 배열에 대해서는 자바 객체를 생성하는 과정을 수행해야 한다. 이 과정은 네이티브 함수에서 배열의 반환과 동일하다. 두 번째 단계는 Call<Type>Method를 사용하여 호출하고 반환 값이 있으면 이를 반환 받는 단계이다. 마지막 단계인 그림 8-②는 반환받은 값을 네이티브 함수에서 접근

가능한 형태로 변환한 후, 프락시 함수를 호출한 네이티브 함수로 값을 되돌려 보내는 작업으로 스텁 함수의 매개변수 준비 작업과 유사하다. 자바 메소드의 반환 값이 참조형인 경우, 네이티브 환경에서 이를 변경하였다면 이 또한 자바 객체에 반영되어야 하는데 이는 스텁 함수에서 담당하고 있다.

자바 메소드 - 자바 클래스

```
int javaM() { ... }
```

프락시 함수 - CDT

```
int javaM() { // 네이티브 자바 메소드의 정의부
    // ① 매개변수 준비 - 자바 객체 생성
    j_retVal =
      (*_CEnv)->CallIntMethod(_CEnv, _CObj, _mid_javaM);
    // ② 반환 값을 C 값으로 변환
    return c_retVal;
}
```

네이티브 함수 - 네이티브 클래스

```
int nM1(IntArr arr) { ... = f1 + javaM(); ...}
```

그림 8. 프락시 함수 - 자바 메소드 호출자

4.2 JaC 클래스 번역기

JaC 클래스 번역기는 JaC 전처리기(JaC preprocessor)와 NMA 생성기(NMA Generator)로 나누어진다. JaC 전처리기는 JaC 클래스를 입력으로 받아 순수 자바 클래스와 네이티브 함수를 분리 추출하는 작업을 담당한다. 순수 자바 클래스는 자바 컴파일러인 javac에 의해 클래스 파일로 번역된다. NMA 생성기는 JaC 파싱 과정에서 얻은 네이티브 메소드의 매개변수 정보, 네이티브 언어 종류, 헤더파일 정보, 그리고 클래스 파일을 가지고 네이티브 클래스와 CDT를 생성한다. 그림 9는 JaC 클래스를 입력으로 받아 이러한 과정을 통합적으로 처리하여 주는 jacc (JaC Compiler)의 구조를 보여주고 있다.

JaC 전처리기는 파서와 자바 클래스 및 C/C++로 구현된 네이티브 함수, 각종 부가정보를 추출하는 추출기로 이루어진다. JaC 클래스는 네이티브 메소드가 C/C++ 코드 블록을 가질 수 있다는 것을 제외하면 자바 클래스와 동일하다. 따라서 JaC 파서는 자바 문법을 약간 수정하여 구성하였다. 파서 생성기는 LL(k) 문법을 입력으로 받아 top-down 파서를 생성하는 JavaCC를 사용하였다[9,10]. 파서는 JaC 클래스를 입력으로 받아 추상 구문 트리(Abstract Syntax Tree: AST)를 생성한다. 추출기는 AST를 입력으로 자바 코드, C/C++ 함수를 출력하며 이 과정에서 각종 부가 정보도 함께 얻는다.

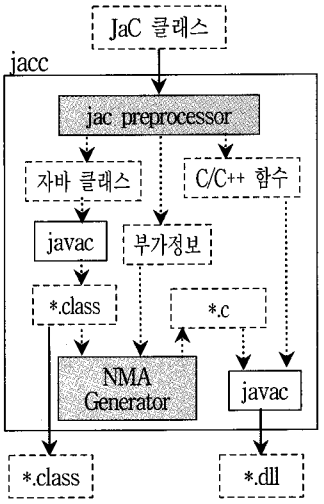

그림 9. jacc의 구조

NMA 생성기는 클래스 파일(*.class)과 부가정보를 입력으로 받아 네이티브 클래스와 CDT로 이루어진 NMA 파일(*.c)을 생성하게 된다. 자바 메소드와 필드에 대한 정보를 추출하기 위해 Apache Jakarta Project 중에 하나인 BCEL(Byte Code Engineering Library)을 사용하였다[11]. 우선 NMA 생성기는 클래스 파일에서 필드와 메소드에 관한 정보를 얻는다. 자바 컴파일러의 디버깅 옵션을 사용하더라도 네이티브 메소드의 매개변수 이름 정보는 클래스 파일에 포함되지 않는다. 따라서 클래스 파일에 포함되어 있지 않은 매개변수 이름, 네이티브 언어 및 헤더파일에 관한 정보 등의 부가 정보는 JaC 전처리기로부터 받는다. 이들 정보를 바탕으로 네이티브 클래스를 구성한다. 네이티브 필드와 네이티브 자바 메소드는 클래스 파일로부터 구성 가능하다. C/C++로 구현된 네이티브 함수 정의는 JaC 전처리기에서 추출된 정보를 가지고 만들어진다. 다음으로 CDT는 네이티브 클래스의 필드 초기화, 자바 객체의 상태 갱신 그리고 자바 메소드 호출, 네이티브 함수 호출을 담당하게 된다. 이를 위해 필드 초기화와 변경된 필드의 갱신 그리고 네이티브 함수를 호출하여 주는 스텁 함수, 자바 메소드 호출을 담당하는 프락시 함수, 그리고 각종 부가 루틴들을 생성한다. 이러한 과정을 거

처 얻어진 NMA 파일(*.c)은 C/C++ 컴파일러에 의해 동적 라이브러리인 DLL로 만들어진다. 최종 출력물인 자바 클래스 파일(*.class)과 NMA 라이브러리(*.dll)는 자바 가상 기계인 java 명령에 의해 실행되어 결과를 얻게 된다.

5. 평가 및 결론

본 장에서는 실제 예제를 통하여 JaC의 사용 편리성을 보여준다. 그리고 본 도구를 사용하여 얻은 실행 속도 측정값을 통하여 본 연구의 실용성을 보임으로서 결론짓고자 한다.

예제 프로그램은 특정 분야 보다는 본 논문에서 제안한 방법이 옳게 작동하는지를 보여주기 위해 다양한 상호 작용을 보여줄 수 있도록 작성하였으며 네이티브 언어로 C++를 사용하였다.

```
public class Simple {
  int f1;
  double[] f2 = {1.1, 2.1, 3.2};
  String msg = "End.";

  @NativeCode(include="iostream", lang="C++")
  native int nM1(int[] arr) {
    int sum = 0;
    f1 = f1 + javaM(20); // 자바 메소드 호출
    for(int i=0; i < arr.length; i++) {
      sum += arr.value[i]; // 매개변수 접근
          std::cout << sum << ", ";
    }
    arr.value[i] = 77; // 배열 원소의 갱신
    std::cout << msg << std::endl;
    return sum;
  }

  native double sumArray(String msg, float fval) {
    double sum=0;
    std::cout << msg << " " << fval << "\n";
    for(int idx = 0; idx < f2.length; idx++)
        sum += f2.value[idx]; // 필드 접근
    return sum;
  }

  int javaM(int arg) { return 10 + arg; }

  public static void main(String[] args) {
    int[] data = {1, 3, 5, 7};
    Simple mm = new Simple();
    mm.f1 = 5;
    System.out.println("Val => " + mm.nM1(data));
    System.out.println("Field f1 => " + mm.f1);
    System.out.println(mm.sumArray("JaC", 3.5f));
    for(int val: data) System.out.print(val + " ");
  }
}
```

그림 10. JaC 클래스 - Simple.jac

그림 10은 네이티브 메소드 nM1()에서 정수형 필드와 배열 매개변수의 접근, 자바 메소드 호출, 배열 갱신을 보여주고 있다. 그리고 네이티브 메소드 sumArray()는 스트링, 실수형 매개변수와 실수형 배열 필드를 접근하고 있다. 그림 11은 그림 10 프로그램을 컴파일하고 실행시키는 과정을 보여주고 있다.

자바와 C/C++의 혼합 프로그래밍으로 인한 JaC의 성능에 대해 알아보기 위해 여러 예제 프로그램을 통하여 검증하여 보았다. 표 5는 Java 6 SDK javac, Visual studio 2008 cl, jacc를 각각 사용하여 Intel E6550 2.33GHz의 Windows Vista 환경에서 다양한 예제 프로그램을 실행해 보았을 때 실행 속도 측정치 나타낸 테이블이다. C 컴파일러에서 최적화 옵션 /O2를 사용하여 측정한 결과이다. 성능 측정에서 JNI를 사용하여 수동으로 작성한 프로그램의 실행 속도 측정치는 jacc와 거의 유사하여 생략하였다. 이는 필드와 메소드 ID 캐싱, 변경되지 않은 객체의 반영 과정 생략, 객체 상태 동기화의 일괄 처리 등 시스템 부담을 최소화하려한 시도의 결과라 보여 진다.

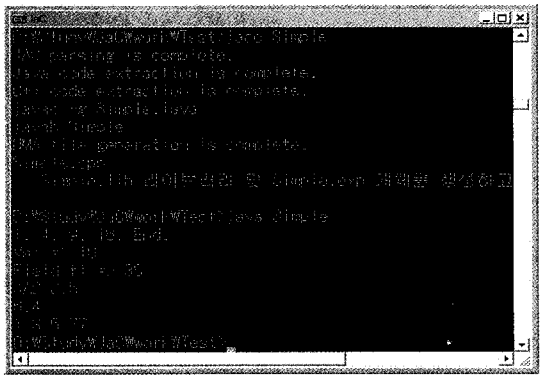

그림 11. Simple.jac의 컴파일과 실행결과

표 5. 실행 시간(단위: 초(sec))

프로그램	javac	cl	jacc
Perfect 40000	4.60	4.39	4.53
Fibonacci 41	2.37	4.75	4.55
Permutation 12	12.2	11.9	13.2
BubbleSort	2.56	0.48	0.68
LoopTest	1.55	0.73	0.86

자바의 경우, 최초의 수행은 JVM의 부팅 시간과 JIT에 따른 번역시간이 추가되므로 자바가 느리다. 따라서 본 측정에서는 이러한 준비 시간이 무시될 수 있는 실행시간을 가지는 예제를 사용하였다.

성능 측정결과 C 언어와 거의 동등한 실행 속도를 보이고 있음을 알 수 있다. 처음 3개의 예제는 순환함수를 사용하고 있는 예제로 자바 언어 와 C 언어 간의 속도 차이는 거의 보이고 있지 않으며 Fibonacci 수열의 경우에는 오히려 자바가 우수함을 보이고 있다. BubbleSort는 100개의 자료를 1000번 반복 수행하였으며, LoopTest는 3중 loop를 사용하여 특정계산을 1억 회 시도하여 측정하였다. 이 두 예제는 모두 C 언어가 빠른 처리속도를 보이고 있다. 이 때 jacc가 C와 비례하여 실행 속도가 빨라짐을 알 수 있다. 반복처리 또는 다중 루프의 경우에는 jacc를 사용함으로서 실행 속도 향상을 이룰 수 있다. 이상의 실험 결과를 토대로 혼합 프로그래밍으로 인한 성능 저하가 발생하지 않고 있음을 알 수 있다. 결론적으로 JaC는 두 언어의 장점인 플랫폼 독립성과 플랫폼 의존성을 모두 수용하면서 성능 저하가 발생하지 않음을 확인하였다.

자바 언어는 플랫폼 독립적이어서 다양한 환경에서 수정 없이 동일한 작업 수행이 가능하다. 네이티브 언어는 하드웨어와 운영체제 등을 직접 조작하는 플랫폼 의존적인 작업이 가능하다. JaC 클래스는 플랫폼 의존적인 작업과 플랫폼 독립적인 작업을 한 개의 클래스에서 모두 가능하도록 하였다. JaC 클래스는 자바 클래스와 네이티브 클래스라는 이중의 개념과, 두 객체 간에 상호 작용을 도와주는 CDT를 두어 제어 및 정보 교환이 가능하도록 하였다. JaC 클래스를 실행 가능한 파일로 번역하여주는 jacc를 구현하고, 다양한 예제 프로그램을 실행시켜 본 연구에서 제안한 방법이 올바르게 작동하고 있음을 보여주었다. 그리고 동일 프로그램에 대해 자바 클래스, C 함수, JaC 클래스로 작성하여 성능 측정치를 보여줌으로써 혼합 프로그래밍으로 인한 성능 저하가 발생하지 않음을 확인하였다. 마지막으로 순수 자바로 작성 가능한 응용 프로그램이라 할지라도 네이티브 메소드로 작성하여 실행 속도 향상을 이룰 수 있는 응용 분야가 있음을 보여주었다.

참 고 문 헌

[1] Ken Arnold, James Gosling and David Holmes, *The Java Programming Language*, 4th Edition, Addition Wesley, 2005.

[2] Java Native Interface Specification. http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html.

[3] Sheng Liang, *The Java Native Interface: Programmer's Guide and Specification*, Addition Wesley, 1999.

[4] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, 2nd Edition, Addison Wesley, 1999.

[5] GlueGen, https//gluegen.dev.java.net

[6] HawtJNI, http://fusesource.org/forge/projects/HAWTJNI.

[7] JNA, http://jna.dev.java.net.

[8] Evgeniy Gabrilovich and Lev Finkelstein, "JNI-C++ integration made easy," *C/C++ Users Journal*, Vol.19, No.1, pp. 10-21, 2001.

[9] JavaCC, https://javacc.dev.java.net.

[10] Viswanathan Kodaganallur, "Incorporating Language Processing into Java Applications: A JavaCC Tutorial," *IEEE Software*, Vol.21, No.4, pp. 70-77, 2004.

[11] BCEL, http://jakarta.apache.org/bcel.

김 상 훈

1986년 2월 동국대학교 이과대학 학사

1989년 2월 동국대학교 컴퓨터공학과 석사

1996년 8월 동국대학교 컴퓨터공학과 박사

1997년 3월~현재 세명대학교 컴퓨터학부 부교수

관심분야 : 프로그래밍언어, 컴파일러, 소프트웨어공학