21-1-R02

가상기계를 위한 네이티브 인터페이스 정의 언어
(Native Interface Definition Language for Virtual Machine)

박지우* · 이창환** · 오세만*
동국대학교 컴퓨터공학과* · 링크젠**
{jojaryong* · yich** · smoh*}@dongguk.edu

요 약

가상기계란 하드웨어로 이루어진 물리적인 시스템과는 달리 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적인 컴퓨터이다. 가상기계에서 실행되는 프로그램은 플랫폼 독립적인 장점이 있지만, 가상기계에서 제공하지 않는 플랫폼 의존적인 기능은 사용할 수 없다. 이와 같은 문제를 해결하기 위해서 가상기계 환경에서는 일반적으로 네이티브 인터페이스를 제공한다. 제공된 네이티브 인터페이스를 통해 구현된 네이티브 함수를 가상기계 환경에서 사용하기 위해서는 네이티브 함수를 위한 정보가 필요하다.

본 논문에서는 네이티브 함수를 위한 정보를 용이하게 생성하기 위해 네이티브 인터페이스 정의 언어와 언어로부터 네이티브 인터페이스 사용에 필요한 정보를 생성하는 컴파일러를 설계 및 구현한다. 구현된 컴파일러의 결과인 네이티브 함수를 위한 정보는 가상기계 환경의 내부 표현으로 생성되며, 생성된 정보는 임베디드 시스템을 위한 가상기계인 EVM에 적용하여 검증한다. 제안된 언어와 컴파일러를 통해 가상기계 개발자가 네이티브 함수를 위한 정보를 직접 작성하는 부담을 줄이고 편의성을 제공할 수 있다.

1. 서 론

다양한 형태의 모바일 기기와 컴퓨터의 등장으로 각각의 단말기에 적합한 서로 다른 프로세서와 운영체제가 사용되고 있다. 다양한 환경은 동일한 작업을 수행하는 프로그램을 각각의 사용자 환경에 맞도록 수정해야 하는 단점을 지닌다. 단점을 극복하기 위해서는 다양한 환경에서도 프로그램의 수정 없이 실행되는 기술을 요구한다. 이에 따른 해결책으로 이종의 장치에 탑재되어 플랫폼 독립성을 제공하는 가상기계가 널리 사용되고 있다.

가상기계는 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적 컴퓨터이다. 따라서 가상기계 환경에서 실행되는 프로그램은 플랫폼 독립적인 장점을 갖는다. 그러나 가상기계에서 제공하지 않는 플랫폼에 의존적인 기능을 사용하는 것이 불가능하다. 가상기계 환경에서는 일반적으로 플랫폼 의존적인 기능을 사용하기 위해 네이티브 인터페이스를 제공한다. 네이티브 언어로 작성된 함수를 사용하는 가상기계 코드의 어셈블(Assemble)과 실행을 위해

서는 네이티브 함수에 대한 추가적인 정보가 필요하다. 네이티브 함수에 대한 정보는 사전에 네이티브 함수 테이블과 함수 원형 파일을 생성하여 가상기계 환경에 제공될 수 있다. 그러나 네이티브 함수 테이블과 함수 원형 파일을 가상기계 개발자가 직접 작성할 경우 문제가 발생할 가능성이 크다.

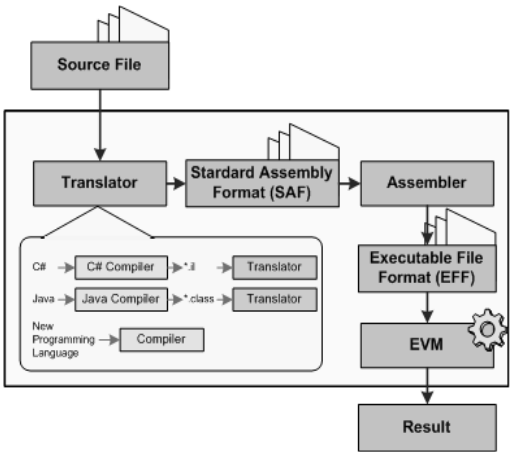
본 논문에서는 네이티브 함수를 위한 정보를 용이하게 생성하기 위해 네이티브 인터페이스 정의 언어와 언어로부터 네이티브 인터페이스 사용에 필요한 정보를 생성하는 컴파일러를 설계 및 구현한다. 네이티브 인터페이스 정의 언어는 명시적으로 함수 정보를 작성할 수 있도록 설계하고 기존 EVM(Embedded Virtual Machine) 환경과의 호환성을 고려한다. 네이티브 정의 언어 컴파일러는 네이티브 정의 언어로 작성된 소스 코드는 입력으로 하여 네이티브 함수 테이블과 네이티브 함수에 대한 원형 파일을 생성한다. 가상기계의 어셈블러, 디스어셈블러, 실행 엔진은 생성된 테이블을 사용하여 네이티브 함수와 관련된 작업을 수행한다. 컴파일러의 결과인 네이티브 함수 테이블과 함수 원형 파일은 임베디드 시스템을 위한 가상기계인 EVM에 적용하여 검증한다. 제안된 언어와 컴파일러의 설계 및 구현을 통해 가상기계 개발자에게 네이티브 함수 테이블과 함수 원형 파일 생성의 편의성을 제공할 수 있다.

본 논문의 2장에서는 본 논문에서 사용한 가상기계인 EVM에 대한 소개와 특징에 대하여 간략히 언급하고, 현재 사용되고 있는 네이티브 인터페이스와 인터페이스 정의 언어에 대해서 정리한다. 3장에서는 네이티브 인터페이스 정의 언어의 역할, 네이티브 인터페이스 정의 언어 문법, 네이티브 인터페이스 정의 언어 컴파일러에 대해 소개한다. 4장 실험 및 결과에서는 실험 데이터를 작성하여 제안한 방법을 EVM에 적용하고 실험하여 결과를 확인한다. 마지막 5장 결론에서는 본 논문에서 제시한 네이티브 인터페이스 정의 언어를 요약하고 향후 연구 방향에 대하여 언급한다.

2. 관련 연구

2.1 EVM(Embedded Virtual Machine)

EVM은 스택 기반 가상기계로서 모바일 디바이스(Mobile Device), 셋톱박스(Set-top Box), 디지털 TV(Digital TV)등에 탑재되어 동적 응용 프로그램을 다운로드하여 실행하는 가상기계 솔루션이다. EVM은 (그림 1)과 같이 크게 변환기, 어셈블러, 가상기계의 세 부분으로 나눌 수 있다.



(그림 1) EVM 시스템 구성도

변환기는 C#이나 자바 등의 고급 프로그래밍 언어로 작성된 프로그램을 가상기계의 어셈블리 포맷인 SAF(Standard Assembly Format)로 번역하고 어셈블러는 SAF를 가상기계에서 실행 가능한 형태인 EFF (Executable File Format) 파일로 변환하며

EVM은 실제 하드웨어에 탑재되어 EFF 파일을 실행하는 가상기계 역할을 한다[1][2].

EVM의 중간 언어인 SIL(Standard Intermediate Language)은 스택 연산, 산술 연산, 흐름 제어 등 총 6개의 카테고리로 분류되는 스택 기반의 명령어 집합으로 언어 독립성과 하드웨어 및 플랫폼 독립성을 갖고 있다. SIL은 다양한 프로그래밍 언어를 수용하기 위해서 바이트코드[3], .NET IL[4] 등 기존의 가상기계 코드들의 분석을 토대로 정의되었다. 또한 프로그램의 확장성을 위해 순차적 언어와 객체지향 언어를 모두 수용할 수 있도록 설계되었다. 각각의 명령어 집합은 EVM에서 실행 될 수 있는 이진 코드와 1:1 대응된다[5][6].

고급 언어로 작성된 코드는 변환기를 통해서 의사 코드와 연산 코드로 구성된 EVM의 어셈블리 포맷인 SAF로 변환된다. 이는 어셈블러에 의해 EFF 형태로 변환되고 시스템의 운영 체제나 구조에 상관없이 EVM에 의해 실행된다. SAF는 임베디드 시스템을 위한 가상기계의 표준 어셈블리 포맷으로 설계되었으며, 클래스 선언 등 특정 작업의 수행을 의미하는 의사 코드와 가상기계에서 실행되는 실제 명령어에 대응되는 연산 코드로 이루어져 있다. 연산 코드는 스택 기반의 명령어 집합이며 특정 프로그래밍 언어에 종속되지 않는 언어 독립성과 하드웨어 및 플랫폼 독립성을 갖고 있다. 따라서 연산 코드의 연상 기호는 특정 하드웨어나 소스 언어에 종속되지 않는 추상적인 형태를 지닌다[7][8].

EFF는 실행에 필요한 모든 정보를 가진 EVM의 실행 파일 포맷으로서 이진 형태의 바이트 스트림으로 구성되어있다. 다양한 프로그래밍 언어를 수용하기 위해서 자바 클래스 파일[3], .NET PE 파일[4] 등 기존에 널리 사용되고 있는 가상기계 실행 파일 포맷들의 분석을 토대로 정의하였다. EFF는 크게 헤더, VM 코드, 메타 데이터의 세 부분으로 구분할 수 있다. 헤더는 해당 파일이 EFF 파일임을 나타내고 프로그램의 버전, 시작점 등과 같은 정보를 가지고 있다. VM 코드는 프로그램에 대한 실질적인 명령어 집합을 포함하고 있다. 메타 데이터는 총 15개의 테이블로 구분되며, 상수 값, 메소드 정보와 같은 실행에 필요한 그 밖의 정보들을 저장하고 있다[9][10][11].

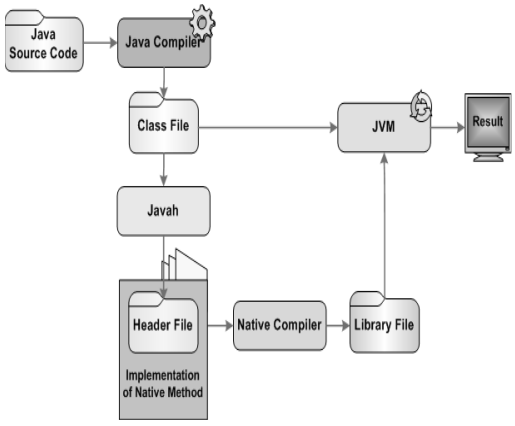
2.2 네이티브 인터페이스

2.2.1 JNI(Java Native Interface)

JNI는 자바에서 제공하는 네이티브 인터페이스다. 자바로 작성된 프로그램은 자바 가상기계(Java Virtual Machine) 상에서 수행된다. 따라서 플랫폼 의존적인 부분을 직접 개발할 수 없다. 이에 대한 해결책으로 자바에서는 네이티브 언어와 연결할 수 있는 방법을 제공하고 있고 이를 네이티브 인터페이스라 지칭한다. 네이티브 인터페이스를 통해 개발자는 플랫폼 의존적인 기능을 구현하거나 이미 다른 언어로 작성된 라이브러리나 프로그램을 사용하여 자바 프로그램을 작성할 수 있다. 또한 프로그램의 성능에 크게 영향을 주는 코드를 네이티브 언어로 작성하여 속도 향상을 꾀할 수 있다[12][13].

네이티브 메소드를 작성하고 실행하는 방법은 (그림 2)에서 볼 수 있듯이 컴파일 과정을 통한 클래스 파일 생성, Javah 유틸리티를 사용하여 헤더 파일 생성, 네이티브 언어로 메소드 구현, 동적 라이브러리 파일 생성, 실행의 5단계로 나눌 수 있다.

우선 자바 프로그래머는 네이티브 메소

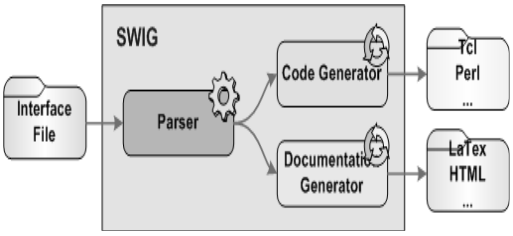


(그림 2) JNI를 통한 네이티브 메소드 구현 과정

드가 포함된 자바 클래스를 만든다. 자바 언어가 아닌 다른 언어로 메소드 구현을 작성한다면, 자바 클래스 안에 메소드를 정의할 때 native 키워드를 포함해야 한다. native 키워드는 함수가 네이티브 언어로 작성된 함수라는 것을 컴파일러에게 알린다. 자바 클래스에서 네이티브 메소드의 선언은 단지 메소드 원형만 있고 구현은 존재하지 않는다. 메소드는 개별적인 네이티브 언어 소스 파일에서 구현된다. 작성한 자바 프로그램을 컴파일(Compile)한 후 javah 유틸리티를 사용하여 네이티브 메소드를 위한 헤더 파일을 생성할 수 있다. 생성된 헤더 파일의 정보를 이용하여 실제 메소드를 C/C++과 같은 네이티브 언어로 구현하여 컴파일러를 사용하여 공유 라이브러리 파일을 생성한다. 자바 가상기계는 클래스 파일과 공유 라이브러리 파일을 사용하여 프로그램을 실행한다[14].

2.2.2 SWIG(Simplified Wrapper and Interface Generator)

언어별로 지원하고 있는 네이티브 인터페이스는 사용 방법이 복잡하고 서로에 대한 호환성을 고려하지 않는다. 따라서 사용자는 언어의 각기 다른 네이티브 인터페이스 사용 방법을 인지하고 있어야 하는 어려움이 있다. SWIG는 C나 C++로 작성된 프로그램과 다른 여러 가지 언어들을 연결시켜서 사용할 수 있도록 도와주는 인터페이스 컴파일러이다[15]. SWIG를 사용하면 서로 다른 언어의 네이티브 함수에 대한 연결을 일관된 방법으로 해결할 수 있다. (그림 3)은 SWIG의 구조를 그림으로 도식화한 것이다.



(그림 3) SWIG의 구조

SWIG의 입력인 인터페이스 파일은 함수 원형과 변수 선언을 포함한다. *%module* 지시자는 SWIG에 의해 생성되는 모듈의 이름을 정의한다. %{, %} 블록은 C 헤더 파일 혹은 추가적인 C 선언과 같은 추가적인 코드를 삽입하기 위한 장소를 규정한다.

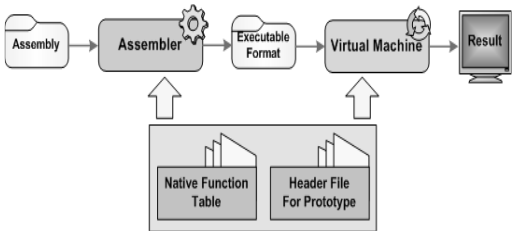
```
/* example.i */
 %module example
 %{
 /* Put header files here or function declarations like below
*/
 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
 %}

 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
```

(그림 4) SWIG의 인터페이스 파일

SWIG 파서는 입력 파일을 구문 분석하여 코드 생성기과 문서 생성기에 전달한다. 각각의 모듈은 특정한 스크립트 언어를 위한 인터페이스와 생성된 인터페이스를 설명하는 문서를 생성한다.

3. 가상기계를 위한 네이티브 인터페이스 정의 언어

가상기계 환경에서 실행되는 프로그램은 플랫폼 독립적인 장점을 갖는다. 그러나 가상기계에서 제공하지 않는 플랫폼에 의존적인 기능을 사용하는 것이 불가능하다. 따라서 가상기계 환경에서는 일반적으로 플랫폼 의존적인 기능을 사용하기 위해 네이티브 인터페이스를 제공한다. (그림 5)는 가상기계의 네이티브 인터페이스를 그림으로 도식화한 것이다.



(그림 5) 가상기계의 네이티브 인터페이스

네이티브 인터페이스를 사용하기 위해서는 가상기계 환경에 네이티브 함수에 대한 추가적인 정보를 내부 표현의 형태로 제공해야 한다.

본 논문에서는 네이티브 함수를 위한 추가적인 정보를 표현하는 네이티브 함수 테이블과 함수 원형 파일을 자동적으로 생성하기 위해 네이티브 함수에 대한 정보를 기술할 수 있는 네이티브 인터페이스 정의 언어를 설계하고 컴파일러를 구현한다.

네이티브 인터페이스 정의 언어는 네이티브 함수를 위한 테이블과 구현하고자 하는 네이티브 함수에 대한 원형을 용이하게 생성하기 위한 인터페이스 정의 언어이다. 일반적인 인터페이스 정의 언어와 비슷하게 함수 집합에 대한 정의만을 허락하며 구현부는 포함할 수 없다. 새로운 언어를 정의하기 위해서는 그 언어에 대한 문법과 컴파일러가 필요하다.

3.1 네이티브 인터페이스 정의 언어의 역할

가상기계에 네이티브 함수로 구현된 기능을 추가하기 위해서는 함수에 대한 정보를 제공하는 테이블과 실제 구현부가 필요하다.

네이티브 함수에 대한 정보는 사전에 네이티브 함수 테이블과 함수 원형 파일을 생성하여 가상기계 환경에 제공될 수 있다. 그러나 네이티브 함수 테이블과 함수 원형 파일을 가상기계 개발자가 직접 작성할 경우 문제가 발생할 수 있다.

첫째로 개발 과정에서 개발자가 처리해야 할 작업량이 늘어나기 때문에 오류 수정이 어렵다. 이진수로 표현되는 정보를 저장하는 네이티브 함수 테이블과 함수 원형 파일을 정해진 규약에 맞게 실수 없이 직접 작성하는 것은 많은 노력이 필요하다.

두 번째는 네이티브 함수 테이블과 함수 원형 파일 사이의 일관성 문제이다. 네이티브 함수 테이블과 함수 원형 파일의 정보는 서로 밀접한 관계를 가지고 있고, 함수 이름 등과 같은 특정 정보는 항상 일치하도록 구성해야 한다. 각각의 개발자가 파일을 직접 수정하면 서로 다른 파일에 대한 일관성을 유지하는데 어려움이 있다.

네이티브 함수 테이블은 함수 인덱스, 이

진수로 표현되는 디스크립터 등의 정보를 가지고 있다. 또한, 테이블은 정해진 구조에 대하여 유연하지 않기 때문에 약간의 오차도 허용하지 않는다. 따라서 네이티브 함수의 수가 증가할수록 오류 없는 테이블을 직접 작성하는 것은 쉽지 않은 작업이다.

네이티브 인터페이스 정의 언어를 도입하면 언급한 문제점을 쉽게 해결할 수 있다. 개발자는 쉽고 명확한 문법으로 정의된 네이티브 인터페이스 정의 언어로 네이티브 함수 정보를 소스 파일로 기술할 수 있다. 네이티브 함수 테이블과 함수 원형 파일은 소스 파일로부터 자동적으로 생성된다. 따라서 개발자는 이진수로 구성된 데이터의 작성과 같은 복잡한 문제점을 고려할 필요가 없고 네이티브 함수 테이블과 함수 원형 파일이 일관성 있는 정보를 유지한 형태로 생성된다. 또한, 가상기계 개발 과정을 단순화하여 프로그램에 대한 오류를 쉽게 수정하고, 오류 자체를 줄일 수 있다.

3.2 네이티브 인터페이스 정의 언어 문법

기존의 인터페이스 정의 언어는 가상기계에 대한 고려가 없기 때문에 가상기계에 그대로 적용하기 어려운 측면이 있고 문법이 복잡하여 컴파일러를 구현하기 힘들다. 특히, 임베디드 가상기계는 제한된 자원을 가진 장치에서 사용되기 때문에 간결한 인터페이스 정의 언어가 필요하다.

본 논문에서는 네이티브 인터페이스를 위한 내부 정보를 최대한 간결한 형태로 기술할 수 있도록 반환값, 매개변수 등 최소한의 정보만을 기술할 수 있는 언어를 정의한다. 이와 같은 설계 목표는 부수적으로 프로그래머가 새로운 언어를 익히는 부담을 줄이는 효과를 얻을 수 있다. (그림 6)은 네이티브 인터페이스 정의 언어의 문법을 EBNF(Extended Backus-Naur Form)의 형태로 표기한 것이다.

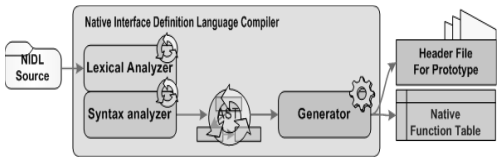
```
<NIDL>      ::= { <ninterface_dcl> }
<ninterface_dcl>
 ::= '.ninterface' <modifiers> <ninterface_name>
      '.bgn' <ninterface_body> '.end'
<ninterface_body>
 ::= <method_dcl> { <method_dcl> }
<method_dcl>
 ::= '.method' <modifiers> <type_specifier>
     <method_name> '(' [<formal_param>] ')'
<formal_param>
 ::= <type_specifier> <formal_param_name>
    { ',' <type_specifier> <formal_param_name> }
<ninterface_name> ::= '$identifier'
<method_name>   ::= '$identifier'
<formal_param_name> ::= '$identifier'
<modifiers> ::= 'public' | 'private' | 'static' |
          'terminal' | 'guarded' |
          'interfacce' | 'concept'
<type_specifier> ::= 'byte' | 'integer' | 'long'
                   | 'float' | 'double' | 'short'
                   | 'char' | 'reference'
                   | 'boolean' | 'void'
```

(그림 6) 네이티브 인터페이스 정의 언어 문법(EBNF)

네이티브 인터페이스 정의 언어는 인터페이스 단위로 구성되어 있다. 따라서 프로그램은 지정어 *.ninterface*를 통해 인터페이스를 선언하는 것으로 시작된다. 하나의 인터페이스 안에는 *.method* 지정어를 사용하여 다수의 함수를 정의할 수 있으며, 함수 정의시 반환값, 매개변수 등의 함수에 대한 정보를 명시할 수 있도록 설계한다. 또한, 접근 수정자와 타입 정보는 가상기계에서 제공하는 종류와 완전히 일치시킨다. 설계된 문법에 따라 네이티브 인터페이스 정의 언어로 작성하는 소스 파일은 함수의 원형만을 정의할 수 있으며, 구현부는 작성할 수 없다.

3.3 네이티브 인터페이스 정의 언어 컴파일러

네이티브 인터페이스 정의 언어 컴파일러는 일반적인 컴파일러와 같이 어휘 분석기, 구문 분석기, 생성기로 구성되어 있다. (그림 7)은 네이티브 인터페이스 정의 언어 컴파일러를 그림으로 도식화한 것이다.



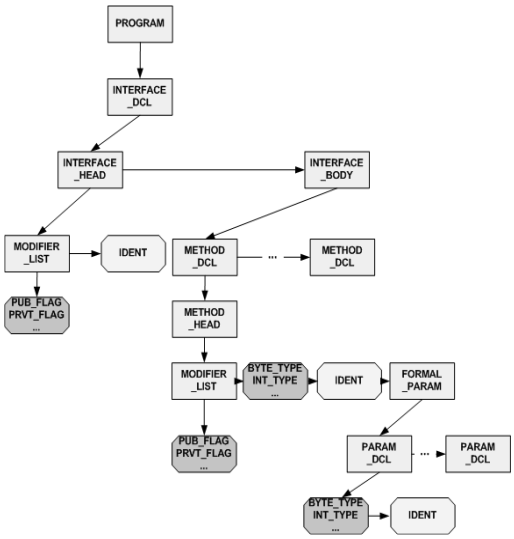
(그림 7) 네이티브 인터페이스 정의 언어 컴파일러

어휘 분석기는 소스 프로그램을 읽어 들여 일련의 토큰을 생성한다. 구문 분석기는 토큰을 입력 받아 소스 프로그램의 에러를 체크하고 올바른 문장에 대해서 구문 구조를 만든다. 구문 구조는 추상 구문 트리의 형태로 생성되며, 생성기는 트리를 운행하여 네이티브 함수 테이블, 네이티브 함수 원형 파일을 생성한다.

파서의 출력인 추상 구문 트리는 (그림 6)의 문법으로 작성한 프로그램의 의미 있는 정보만을 표현하는 트리로써, (그림 8)과 같은 형태로 설계한다.

생성기는 네이티브 인터페이스 컴파일러의 출력인 네이티브 함수 테이블, 네이티브 함수에 대한 원형 파일을 생성하며 테이블 정보는 (표 1)과 같이 네이티브 함수에 대한 인덱스, 이름, 디스크립터와 링크로 구성된다.

네이티브 함수 원형 파일은 네이티브 함수 구현시 필요한 네이티브 함수에 대한 원형을 저장한다. (그림 9)는 네이티브 함수 원형을 위한 헤더 파일의 구조를 나타낸 것이다.



(그림 8) 네이티브 인터페이스 정의 언어의 추상 구문 트리

```
#ifndef __NATIVE_FUNCTION_H__
#define __NATIVE_FUNCTION_H__

#ifdef __cplusplus
extern "C" {
#endif //__cplusplus

#include "NativeInterface.h"

    Native Function Prototypes
    void ENF_<Interface_Name>_<Function_Name>
        (pEVM pVM, int *error)
                    ⋮

#ifdef __cplusplus
}
#endif //__cplusplus

#endif //__NATIVE_FUNCTION_H__
```

(그림 9) 네이티브 함수 원형을 위한 헤더 파일의 구조

(표 1) 네이티브 함수 테이블

항 목		설 명
Index		네이티브 함수의 인덱스
Name		네이티브 함수의 이름
Descriptor	Flag	필드와 함수를 구분
	Length	속성의 길이
	Attribute	네이티브 함수의 매개변수 및 반환 값
Link		네이티브 함수의 구현 부분과 연결

헤더 파일은 *NativeInterface.h* 파일을 포함하고 있다. 함수 이름은 가독성을 높이고 함수 이름 충돌을 피하기 위해 네이티브 함수임을 나타내는 문자 *ENF_*와 함수를 기능별로 구분하는 모듈 이름, 해당 함수가 처리하는 작업을 명시적으로 나타내는 명칭으로 구성된다. 매개변수는 가상기계의 자료 구조와 오류 정보를 전달하기 위한 변수로 구성되어 있다. 네이티브 함수 구현부는 이 파일을 내부적으로 포함하여 작성된다.

4. 실험 및 결과

본 논문에서 제안한 네이티브 인터페이스 정의 언어는 컴파일러의 출력인 네이티브 함수 테이블과 함수 원형 파일을 임베디드 시스템을 위한 가상기계인 EVM에 적용하여 검증한다. 네이티브 인터페이스 정의 언어 컴파일러는 ANSI-C 호환 컴파일러인 Microsoft Visual C++ 2005를 사용하여 구현하였다. 파서는 차후 문법의 수정 혹은 기능 추가를 고려하여 확장성이 뛰어난 LR(Left to right scanning / Right parse) 구문 분석기의 형태로 구현하였으며, LR 구문 분석기 구현 시 필요한 파싱 테이블은 동국-PGS(Dongguk-Parser Generating System)를 통해 생성하였다. 실험은 펜티엄4 2.0 프로세서와 1기가의 메모리를 갖는 IBM 호환 컴퓨터와 운영체제로서 Microsoft Windows XP를 사용하였다.

가상기계에 정수, 실수, 문자 등의 입·출력 기능을 추가하기 위해 네이티브 인터페이스 정의 언어를 사용하여 표준 입·출력 함수의 정보를 (그림 10)과 같이 기술하였다.

```
.interface public StdIO
.bgn
  .method public void PrintInt(integer i)
  .method public void PrintFloat(float f)
  .method public void PrintChar(char c)
  .method public void PrintLn()
  .method public integer ReadInt()
  .method public float ReadFloat()
  .method public char ReadChar()
.end
```

(그림 10) 표준 입·출력 함수에 대한 인터페이스 정의 파일

```
#ifndef __NATIVE_FUNCTION_H__
#define __NATIVE_FUNCTION_H__
#ifdef __cplusplus
extern "C" {
#endif //__cplusplus

#include "ExeEngine/NativeInterface/NativeInterface.h"
enum nativeFunctionIndex {
   iENF_StartIndex = 0xFFFEFFFF,
   iENF_StdIO_PrintInt, iENF_StdIO_PrintFloat,
   iENF_StdIO_PrintChar, iENF_StdIO_PrintLn,
   iENF_StdIO_ReadInt, iENF_StdIO_ReadFloat,
   iENF_StdIO_ReadChar  };

NITable nativeTable = { 7,
  {{iENF_StdIO_PrintInt, "ENF_StdIO_PrintInt",
    {0x0000, 4, {0x0000, 0x5300}}, ENF_StdIO_PrintInt},
   {iENF_StdIO_PrintFloat, "ENF_StdIO_PrintFloat",
    {0x0000, 4, {0x0000, 0x5400}}, ENF_StdIO_PrintFloat},
   {iENF_StdIO_PrintChar, "ENF_StdIO_PrintChar",
    {0x0000, 4, {0x0000, 0x5100}}, ENF_StdIO_PrintChar},
   {iENF_StdIO_PrintLn, "ENF_StdIO_PrintLn",
    {0x0000, 2, {0x0000}}, ENF_StdIO_PrintLn},
   {iENF_StdIO_ReadInt, "ENF_StdIO_ReadInt",
    {0x0000, 2, {0x5300}}, ENF_StdIO_ReadInt},
   {iENF_StdIO_ReadFloat, "ENF_StdIO_ReadFloat",
    {0x0000, 2, {0x5400}}, ENF_StdIO_ReadFloat},
   {iENF_StdIO_ReadChar, "ENF_StdIO_ReadChar",
    {0x0000, 2, {0x5100}}, ENF_StdIO_ReadChar}    }};

#ifdef __cplusplus
}
#endif //__cplusplus
#endif //__NATIVE_INTERFACE_TBL
```

(그림 11) 네이티브 함수 테이블 파일

네이티브 인터페이스 정의 언어 컴파일러를 사용하여 작성한 소스 코드를 컴파일한 결과 네이티브 함수 테이블과 네이티브 함수 원형 파일이 생성되었다.

```
#ifndef __NATIVE_FUNCTION_H__
#define __NATIVE_FUNCTION_H__
#ifdef __cplusplus
extern "C" {
#endif //__cplusplus

#include "ExeEngine/NativeInterface/NativeInterface.h"
void ENF_StdIO_PrintInt(pEVM pVM, int *error);
void ENF_StdIO_PrintFloat(pEVM pVM, int *error);
void ENF_StdIO_PrintChar(pEVM pVM, int *error);
void ENF_StdIO_PrintLn(pEVM pVM, int *error);
void ENF_StdIO_ReadInt(pEVM pVM, int *error);
void ENF_StdIO_ReadFloat(pEVM pVM, int *error);
void ENF_StdIO_ReadChar(pEVM pVM, int *error);

#ifdef __cplusplus
}
#endif //__cplusplus
#endif //__NATIVE_FUNCTION_H__
```

(그림 12) 네이티브 함수 원형 파일

네이티브 함수 테이블에서는 열거형을 사용하여 인덱스를 결정하게 되며, 내부적으로 복잡한 필드 구성을 갖는 테이블 자료 구조를 자동적으로 생성한다.

네이티브 인터페이스 정의 언어 컴파일러의 출력으로 생성된 네이티브 함수 테이블과 함수 원형 파일을 검증하기 위해 EVM에 적용하고 다양한 프로그램을 통해 네이티브 함수의 작동을 검증하였다. 본 논문에는 시스템 의존적인 기능인 표준 입·출력에 관한 함수를 사용하는 계산기 프로그램을 수록하였다. 작성된 프로그램은 사칙 연산과 나머지 연산 등 다양한 연산 기능을 지원한다. 프로그램 및 실행 결과는 (그림 13), (그림 14)와 같다.

계산기 프로그램은 표준 입력을 통해 수행할 연산과 피연산자를 결정하고, 표준 출력으로 결과를 내보낸다.

5. 결론 및 향후 연구

가상기계 환경에서 실행되는 프로그램은 플랫폼 독립적인 장점이 있지만, 가상기계

```
.class public CalculatingMachine
.bgn
   .method public void main()
   .bgn
   .locals (integer choice, operand1, operand2)
      $$3:  ldc.i        1
                  ⋮
            calli ENF_StdIO_ReadInt
            str.i choice
            ldl.i choice
            ldc.i 5
            eq.i
            brt $$1
                  ⋮
            ldc.i 1
            eq.i
            brf $$2
            add.i
                  ⋮
   .end
.end
```

(그림 13) 계산기 프로그램

```
>>>>>> Starting program <<<<<<
   1  Addition
   2  Subtraction
   3  Multiplication
   4  Division
   5  Modulation
   6  Quit
Select Menu 4
Operand 2048
Operand 256
Result is 8
```

(그림 14) 계산기 프로그램 실행 결과

에서 제공하지 않는 플랫폼 의존적인 기능은 사용할 수 없다. 이와 같은 문제를 해결하기 위해서 가상기계 환경에서는 일반적으로 네이티브 인터페이스를 제공한다. 네이티브 인터페이스를 사용하면, 가상기계에서 네이티브 코드로 작성된 프로그램을 사용할 수 있다. 네이티브 코드로 작성된 프로그램 혹은 함수는 가상기계에 네이티브 함수를 위한 정보를 제공해야한다. 네이티브 함수에 대한 정보는 일반적으로 네이티브 함수 테이블과 함수 원형 파일을 통해 제공된다.

본 논문에서는 네이티브 함수 테이블과 함수 원형 파일을 자동적으로 생성하기 위해 네이티브 인터페이스 정의 언어와 언어로부터 네이티브 인터페이스 사용에 필요한 정보를 생성하는 컴파일러를 설계 및 구현하였다. 네이티브 인터페이스 정의 언어는 기존 EVM 환경과의 호환성을 고려하고, 명시적으로 함수 정보를 작성할 수 있도록 설계하였다. 컴파일러의 결과인 네이티브 함수 테이블과 함수 원형 파일은 임베디드 시스템을 위한 가상기계인 EVM에 적용하여 검증하였고, 예제 프로그램을 작성하여 정확한 동작 여부를 확인하였다. 제안된 언어와 컴파일러의 설계 및 구현을 통해 가상기계 개발자에게 네이티브 함수 테이블과 함수 원형 파일 생성의 편의성을 제공할 수 있었다. 또한, 네이티브 함수 테이블과 함수 원형 파일을 가상기계 개발자가 직접 작성할 경우 발생하는 문제점을 해결하여 개발 과정을 단순화하고 각각의 파일에 대한 정보의 일관성을 유지할 수 있었다.

향후에는 네이티브 함수를 사용하는 가상기계 코드를 네이티브 함수에 대한 정보와 독립적으로 어셈블할 수 있는 방법에 대한 연구가 필요하다. 추가적으로, 네이티브 함수에 대한 의사 코드를 추가하여 어셈블러와 네이티브 정의 언어 컴파일러를 통합하는 방안에 대한 연구도 고려할 수 있다.

참 고 문 헌

[1] 오세만, 이양선, 고광만, “임베디드 시스템을 위한 가상기계의 설계 및 구현”, 멀티미디어학회 논문지, 제 8권, 제 9호, pp.1282-1291, 2005.

[2] 손윤식, 오세만, “실행 파일 포맷 생성기의 설계 및 구현”, 한국정보처리학회 추계학술발표대회 논문집, 제 11권, 제 2호, pp.623-626, 2004.

[3] 전병준, 이창환, 오세만, “퍼베이시브 컴퓨팅을 위한 가상기계의 어셈블러”, 한국정보처리학회 추계학술발표대회 논문집, 제 13권, 제 2호, pp.589-592, 2006.

[4] 최유리, 이창환, 오세만, “퍼베이시브 컴퓨팅을 위한 가상기계의 디스어셈블러”, 한국정보처리학회 추계학술발표대회 논문집, 제 13권, 제 2호, pp.585-588, 2006.

[5] 박지우, 이창환, 오세만, “퍼베이시브 컴퓨팅을 위한 가상 기계의 실행 엔진”, 한국정보처리학회 추계학술발표대회 논문집, 제 13권, 제 2호, pp.581-584, 2006.

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

[7] MSIL Instruction Set Specification, Microsoft Corporation, 2000.

[8] SIL Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

[9] SAF Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

[10] 정한종, 임베디드 가상기계를 위한 실행 파일 포맷, 동국대학교 석사학위논문, 2004.

[11] EFF Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

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

[13] Rob Gordon, Essential JNI: Java Native Interface, Prentice Hall, 1998.

[14] Mary Campione, Kathy Walrath and Alison Huml, The Java Tutorial Continued: The Rest of the JDK,

Addison Wesley, 1998.
[15] David M. Beazley, SWIG Users Manual, http://www.swig.org/Doc1.1/HTML/Contents.html, 1997.

박 지 우

1999년~2004년 동국대학교 전자공학과(학사)
2006년~현재 동국대학교 컴퓨터공학과(석사과정)
관심분야 : 프로그래밍 언어, 컴파일러, 가상기계

이 창 환

1994년~1998년 동국대학교 컴퓨터공학과(학사)
1998년~2000년 동국대학교 컴퓨터공학과(석사)
2000년~2003년 동국대학교 컴퓨터공학과(박사)
2006년~현재 (주)링크젠 책임연구원
2007년~현재 동국대학교 산업기술연구원 겸임교수
관심분야 : 프로그래밍 언어, 컴파일러, 내장형시스템

오 세 만

1993년~1999년 동국대학교 컴퓨터공학과 대학원 학과장
2001년~2003년 한국정보과학회 프로그래밍언어연구회 위원장
2004년~2005년 한국정보처리학회 게임연구회 위원장
1985년~현재 동국대학교 컴퓨터공학과 교수
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅