네이티브 코드 연결 기법에 관한 연구

유재민° 고광만
상지대학교 컴퓨터정보공학부
e-mail : {comike°, kkman}@mail.sangji.ac.kr

A Study on the Native Code Connection Technique

Jaemin Yu° , Kwangman Ko
School of Computer and Information Engineering, SangJi University

요 약

최근 다양한 모바일 기기가 등장하면서 플랫폼에 독립적인 응용 프로그래밍 실행 환경을 구축하기 위해 가상기계(Virtual Machine)와 가상기계 기반의 언어가 등장하고 있다. 가상기계 환경에서 수행되는 응용 프로그램의 원활한 수행을 위해서는 입출력, 시스템 함수 호출 등에 대해서는 플랫폼에 의존적인 네이티브 코드를 적절한 방식으로 연결해주어야 한다. 본 논문에서는 기존의 가상기계들이 특정 플랫폼에 탑재되는 방식 및 네이티브 코드 연결 방식을 고찰한 후 본 연구팀에서 현재 개발중인 임베디드 시스템을 위한 가상기계(EVM)에 적용하여 보다 효과적인 연결 기법을 제시하고자 한다.

1. 서 론

최근 급속한 하드웨어 발전 및 다양한 종류의 모바일 장치들이 등장하고 이러한 장치들을 제어하기 위한 다양한 실행 환경이 등장하고 있다. 이러한 환경 속에서 개발자들은 언어뿐만 아니라 자신의 프로그램이 구동할 수 있는 플랫폼까지 고려해야 한다는 부담을 갖는다. 가상기계는 플랫폼에 독립적인 응용 프로그래밍 실행 환경을 지원하고 가상기계 환경에서 개발된 응용 프로그램은 플랫폼에 독립적인 특성을 갖는다[5].

가상기계를 기반으로수행되는 응용 프로그램은 플랫폼 독립적인 특성을 갖지만 가상기계의 다양한 요소에 대해서는 의존적인 성질을 갖는다. 즉 플랫폼 환경에서 작동하고 있는 가상기계라는 응용 프로그램의 한계를 가지고 있다. 자바에서는 자바 가상기계(JVM) 환경에서 바이트코드가 소스 프로그램의 의미를 수행하며 수행 중에 표준 입출력, 다양한 시스템 호출 등에 대해서는 제약을 받는다. 이러한 중간언어들이 시스템 함수를 호출하기 위해서는 가상기계로부터 제약을 받지 않으며 플랫폼에 종속적인 언어로 구현된 네이티브 코드를 가상기계에 내장 함으로서 시스템 함수를 호출할 수 있게 된다[5]. 본 논문에서는 소규모 장치 및 모바일 장치 등에 탑재되고 있는 KVM, Waba VM의 탑재 기법 및 네이티브 코드 연결 기법을 고찰한 후 본 연구팀에서 현재 개발중인 임베디드 시스템을 위한 가상기계(EVM)에 적용하여 보다 효과적인 연결 기법을 제시하고자 한다.

2. 기반 연구

2.1 가상기계 탑재 기법

가상기계는 중간언어를 읽어 들여 다양한 검증 동작을 수행한 후 메모리에 적재하는 로더/링커 부분, 플랫폼에 독립적인 부분인 vmCore 부분으로서 메모리에 탑재되어 있는 정보를 읽어 들여 실제 실행 결과를 생성하는 실행 엔진 부분(interpreter), 가상기계를 특정 플랫폼에 탑재하여 플랫폼 자원을 활용할 수 있도록 하는 어댑터 부분으로 크게 구성되어 있다[3][7][8]. 가상기계 탑재는 어댑터를 구현하는 것으로 어댑터는 크게 3가지 부분으로 나누어서 구현할 수 있다. 첫째, vmCore에서 플랫폼에 독립적이기 위해 별도로 요구하는 자료형을 맞추어주는 정의 부분, 둘째, vmCore의 입출력과 실행을 위한 인터페이스를 제공하는 주 어댑터 부분, 셋째, 운영체제와 가상기계간에 실질적인 인터페이스를 담당하는 네이티브 코드 부분으로 구성되어 있으며 어댑터 구현의 대부분은 네이티브 코드 부분의 구현이 가장 큰 비중을 차지한다

2.2 네이티브 코드(native code)

네이티브 코드는 가상기계에서 사용할 언어가 제공할 API와 밀접한 관계를 가지고 있다. 이 API들은 프로그램이 실행하기 위해서 필수적인 사항이며 표준입출력 같은 플랫폼에 종속적인 기능을 사용하기 위해서는 해당 API가 가상함수로 제작되어 시스템의 해당 기능을 호출하는 구조로 만들어 져야 한다[3].

시스템의 해당 기능의 호출이라 함은 가상기계의 인터프리터가 수행 중에 내부 구현이 없는 가상 함수를 만나게 되면 가상기계 내부에 있는 네이티브 코드중에서 해당 기능을 수행 하 수 있는 네이티브 코드를 호출하여 인터프리터와는 별개로 실행하여 결과만 돌려주는 형식으로대체하는 형식으로 작동하게 된다. 네이티브 코드는 가상기계를 구현 할 때 사용한 언어를 이용하여 구현하는 것이 보편적이며 가상함수들과는 1:1로 매핑되며 해당 기능들을 수행할 수 있는 여러 개의 함수들로 구성된다. 가상기계상의 언어가 가상기계에서 작동하면서 시스템 호출 등 플랫폼에 종속적인 작업을 수행하기 위해서는 네이티브 코드는 필수적이며 가상기계를 특정 플랫폼에 탑재하기 위해서는 해당

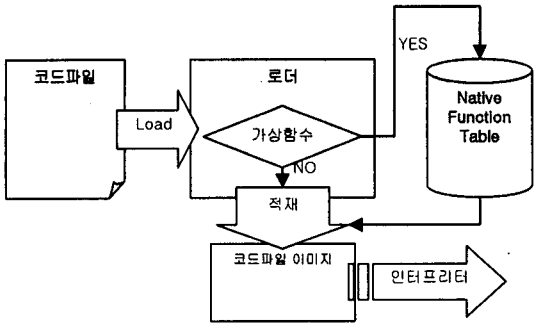
본 논문은 한국과학재단의 특정기초연구(과제번호: R01-2002-000-00041-0)지원에 의한 것임.

플랫폼에 맞추어서 네이티브 코드를 작성해야 한다.

3. 네이티브 코드의 인터페이스

3.1 네이티브 코드의 구성 요소

가상기계의 로더에서 중간 파일로부터 실행 정보를 메모리에 적재할 때 가상함수에 대해서는 파일에 있는 코드 대신 네이티브 코드에 대한 네이티브 함수의 포인터를 적재하며 가상함수와 네이티브 함수의 매핑 관계는 네이티브 함수 테이블에 명시 되어있다[3][7].



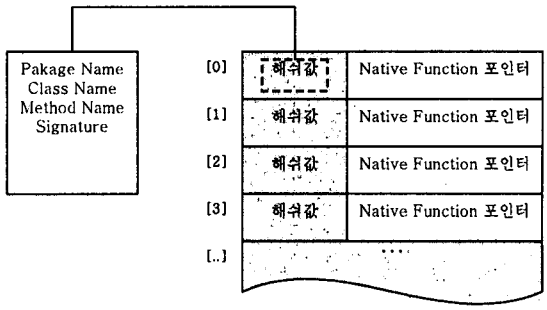
[그림 1] 로더와 네이티브 함수 테이블

네이티브 함수 테이블을 통해 가상함수와 1:1의 기능을 매핑하는 함수를 네이티브 함수라 하며 네이티브 함수에서 실질적인 시스템 호출이 일어나게 되는 것으로 가상기계의 네이티브 코드 실행을 위한 부분은 네이티브 함수 테이블과 네이티브 함수 부분으로 나누어 진다.

3.2 네이티브 함수 테이블(Native Function Table)

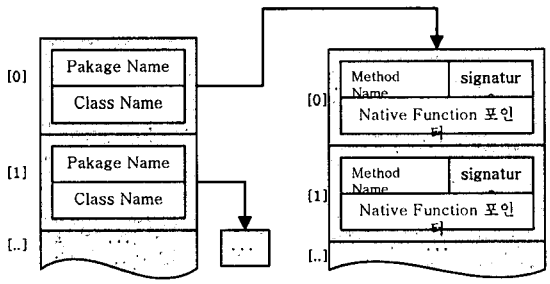
네이티브 함수 테이블은 API상에 있는 가상함수가 어떤 네이티브 함수와 매핑을 이루는지를 명시하고 있는 테이블로서 로더에서 중간코드를 로딩할 경우 사용되는 테이블이다. 테이블을 구성하고 있는 요소는 가상함수를 구별하기 위한 Package name, Class name, signature, method name 과 네이티브 함수를 가리키는 함수 포인터로 구성되어있다. 구현 방식은 KVM 에서는 배열을 이용하고 있으며 WABA VM에서는 해쉬 값과 함수의 포인터로 구성되어 있는 배열을 이용하고 있다. 두 가지 방식의 차이점은 전자의 경우는 테이블의 메모리 점유율이 높은 단점은 있지만 API에 가상함수를 추가하여 API를 확장 할 경우 네이티브 함수 테이블의 재구성이 손쉽다는 점과 속도가 빠르다는 이점이 있다. 후자의 경우는 메모리 점유율 면에서는 우월하지만 API의 확장이 어려운 단점이 있다

WABA VM[7]의 네이티브 함수 테이블은 메모리가 부족한 장치에 적합하게 적은 양의 메모리를 점유 하도록 설계되어 있다. 어떠한 가상함수인지 파악하기 위한 정보들을 전부 저장하지 않고 이 자료들을 가지고 해쉬 값을 구한 다음에 해쉬 값을 해쉬 테이블의 인덱스로 사용하는 것이 아니라 배열에 해쉬값을 넣어 배열을 검색 하는 방식을 사용하고 있다. 검색이 이루어지는 동한 매번 해쉬 값을 구하는 작업 및 모든 가상함수의 리스트가 열거되어 있어서 시간적으로는 테이블의 효율이 상당히 떨어지는 단점이 있다. 또한 테이블에 함수를 추가하기 위해서는 필요한 정보들을 모두 모아서 계산을 해서 테이블을 재구성해야 한다는 단점을 가지고 있다



[그림 2] WABA VM의 네이티브 함수 테이블 구현

KVM[3]의 네이티브 함수 테이블은 클래스의 이름을 가지고 있는 배열이 별도로 존재하며 각각의 구성 요소는 자기자신의 가상함수(메소드)의 배열을 링크하고 있는 구조로 만들어져 있다. 우선 모든 정보가 테이블에 들어 있으므로 테이블의 크기는 조금 큰 편이라고 할 수 있으나 검색속도는 WABA VM과 같은 계산 과정도 필요 없으며 클래스 이름으로 먼저 검색을 하고 메소드 이름으로 검색을 하기 때문에 검색 속도는 WABA VM에 비해서 빠르다. 또한 API 확장으로 인하여 네이티브 함수를 추가해야 할 경우에도 단순하게 명칭을 가지고 구성 할 수 있으므로 테이블의 확장 및 재구성이 용이하다



[그림 3] KVM의 네이티브 테이블 구조

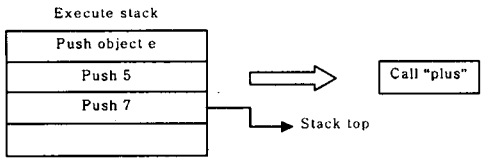
3.3 네이티브 함수(Native Function)

네이티브 함수는 가상기계상의 언어에서 가상 함수를 대신하는 가상기계 내부에 있는 함수로서 꼭 시스템 호출뿐만 아니라 다른 나머지 함수들도 가상 함수를 통해 네이티브 함수를 호출하는 형태로 구현될 수 있으며 네이티브 함수에 의존도가 높아 질수록 가상기계의 속도는 증가하지만 크기 커지게 된다. 따라서 반드시 필요한 시스템 호출 함수만 네이티브 함수로 구현하는 것이 일반적이며 이를 통하여 가상기계가 특정 플랫폼에 탑재 될 수 있는 것이다. 네이티브 함수의 인터페이스는 가상함수에서 전달받는 매개

변수들과 객체를 인터프리터의 실행 스택에 저장하고 나서 네이티브 함수를 호출하게 되고 네이티브 함수가 실행되면서 실행 스택으로부터 가상함수의 매개변수와 객체를 팝하여 해당 작업을 수행한 뒤 리턴 값을 다시 실행 스택에 저장함으로서 작업을 완료하게 된다.
네이티브 코드로 구현해야 할 부분은 프로그램이 수행되기 위해서 시스템에 종속적일 수 밖에 없는 부분으로서 기본적인 입출력, Network, GUI , 디바이스 요청 같은 부분을 수행해야 하는 부분이다. 이러한 기능을 해야 하는 API들은 네이티브 함수를 호출하는 가상함수로 구현되며 Linux에 탑재를 위한 네이티브 코드 구현에서 사용해야 할 요소들을 정의하였다. Linux환경이 PC나 PDA처럼 GUI를 제공 할 경우는 GTK+ 이용 하여 네이티브 코드를 구성하여 그래픽 관련 API를 제공 할 수 있도록 하는 것을 기본 방침으로 삼고 있지만 만일 GTK+ 가 아닌 QT 같은 라이브러리를 이용하여 GUI를 제공하는 시스템일 경우에는 네이티브 코드 부분에서 그래픽 관련 코드 부분을 QT를 해당 라이브러리를 이용하여 제작한다.

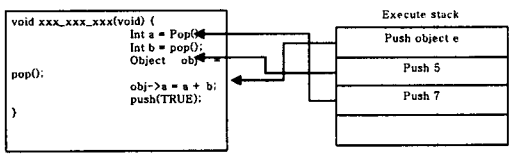
3.4 네이티브 함수의 구현

네이티브 함수의 인터페이스는 기본적으로 인터프리터가 호출하여 실행되고 인터프리터의 실행 스택을 통해서 데이터를 얻고 데이터를 갱신 하는 작업을 하게 된다. 그러나 가상함수의 상태에 따라서 실행 스택에 적재하는 방식이 차이 때문에 네이티브 함수의 작동 방식도 약간의 차이점을 가지고 있다. 일반적인 가상 함수에서는 인터프리터에서 실행 중 [그림 4]처럼 매개변수와 객체를 적재한 후 네이티브 함수를 호출한다.



[그림 4] non-static일 경우 실행 스택의 상태

네이티브 함수가 호출이 되면 팝 동작을 3번하여 가상함수의 매개변수와 객체 자신을 얻어 해당 작업을 수행하게 되며 필드 값의 접근에서는 마지막으로 팝하여 객체 데이터를 통해서 객체의 필드에 접근한다. 이와 같이 실행 스택으로부터 얻은 데이터를 가지고 모든 작업을 마치고 결과 데이터를 가상함수에서 요구하던 반환값과 동일한 자료형으로 실행 스택에 저장하게 되면 네이티브 함수의 호출이 종료된다.



[그림 5] 네이티브 함수의 실행

정적 가상 함수일 경우는 실행 스택에 데이터가 저장되며 네이티브 함수가 비정적(non-static)일 경우와 마찬가지로 팝 동작을 매개변수의 개수만큼 수행하며 가상함수의 데이터를 얻어 해당 작업을 수행 한 다음에 가상함수의 리턴 값과 동일한 자료 형으로 실행 스택에 저장해 줌으로서 작업을 종료하게 된다.

4. 결론 및 향후 연구

가상기계 개발시에 요구되는 네이티브 코드 연결 기법을 KVM, WABA VM에서 세부적으로 고찰하였으며 본 연구팀에서 현재 개발중인 임베디드 시스템을 위한 가상기계(EVM)에 적용하여 보다 효과적인 연결 기법을 제시하는 연구를 진행하였다. 현재까지 연구된 내용에 대한 장단점에 대한 보완 연구를 진행하여 최적의 기법을 새로운 가상기계 개발에 적용하는 연구를 진행할 예정이다.

참고문헌

[1] Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification 2nd edition, Addision-Wesley, 1999.
[2] Bill Blunden, Virtual Machine Design and Implementation in C/C++, Wordware Publishing,Inc., 2002.
[3] Sun Microsystems, The K Virtual Machine(KVM) White Paper. Technical report, Sun Microsystems, 1999.
[4] John R. Levine, Linkers and Loaders, Morgan Kaufmann Publishers, 2000.
[5] Nik Shaylor, Douglas N. Simon, William R. Bush, A Java Virtual Machine Architecture for Very Small Devices, In the Proceddings of ACM SIGPLAN Conferences on Languages, Compilers, and Tools Embedded Systems 2003(LCTES '03), PP. 34-41, ACM Press, 2003.
[6]W. Paugh, Compressing Java class files, In the Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation(PLDI) '99, May, 1999. pp.247-258.
[7] Waba Programming Platform
http://www.wabasoft.com
[8] Joeq Virtual Machine
http://sourceforge.net/projects/jeoq
http://www.stanford.edu/~jwhaley