가상기계를 위한 네이티브 함수 연결 기법에 관한 연구

고 광 만†

요 약

본 논문에서는 KVM, WabaVM의 네이티브 함수 연결 기법을 기반으로 보다 새로운 네이티브 함수 연결을 위한 테이블을 설계하고 구현하였다. 이를 위해 어댑터 모델을 설계하고 기존 네이티브 함수 연결 테이블을 구현하였으며 실제로 구현된 결과에 대한 다양한 실험 결과를 제시하였다.

키워드 : 가상 기계, 어댑터, 네이티브 함수, 네이티브 함수 테이블

A Study on the Native Function Connection Technique for the Virtual Machines

Ko Kwang Man†

ABSTRACT

This paper presents of the native function connection technique for the embedded virtual machines, base on the native function connection methods of the virtual machines such as KVM, WabaVM. For this goals, we designs the adapter model and then implements the new native function table for the native function connection. And we presents the variety experiment and analysis results using the implemented technique.

Key Words : Virtual Machine, Adapter, Native Function, Native Function Table

1. 서 론

가상기계(virtual machine)란 프로세서, 운영체제 등이 바뀌더라도 소스 프로그램을 변경하지 않고 사용할 수 있는 기술로 특히, 임베디드 시스템을 위한 가상기계 기술은 모바일 디바이스, 디지털 TV 등에 탑재할 수 있는 핵심 기술로 다운로드 솔루션에서는 꼭 필요한 소프트웨어 기술이다. 응용 프로그램의 개발 및 실행 방법은 크게 네이티브 애플리케이션과 가상기계 애플리케이션으로 나눌 수 있으며 전자는 이제까지 사용하던 방법으로 실행속도 면에서는 탁월한 장점을 갖는다. 그러나 플랫폼이 변경되면 모든 응용 프로그램을 변경해야 할 뿐만 아니라 심지어는 사용할 수 없게 된다. 이러한 단점을 극복하기 위해서 가상기계를 탑재하여 응용 프로그램을 실행시켜주는 가상기계 솔루션이 등장하였으며 이를 지원하기 위한 다양한 연구가 진행중이다.

가상기계 환경을 기반으로 수행되는 응용 프로그램은 플랫폼 독립적인 특성을 갖지만 가상기계의 다양한 요소에 대해서는 의존적인 성질을 갖는다. 즉, 플랫폼 환경에서 작동하고 있는 가상기계라는 응용 프로그램의 한계를 가지고 있다. 가상기계를 특정 플랫폼에 탑재하는 것은 가상기계라는 프로그램이 특정 플랫폼의 자원을 충분히 활용할 수 있도록 메모리 크기, 자료형, 입/출력 장치의 인터페이스 등을 고려하여 가상기계를 재구성하는 것이다[3][10][12].

가상기계 구현은 가상기계의 핵심 내부 동작을 구현하는 부분과 특정 플랫폼과 인터페이스를 담당하는 어댑터 부분으로 크게 구분된다. 따라서 가상기계의 핵심 부분은 플랫폼에 독립적인 부분으로서 특정 플랫폼에 탑재하더라도 코드의 수정 없이 구현된다. 어댑터는 가상기계를 이용하여 응용 프로그램을 원활히 수행할 수 있도록 플랫폼의 특성에 의존하는 입출력 함수, 시스템 호출 함수 등과 같은 플랫폼에 의존적인 네이티브 함수를 적절한 방식으로 응용 프로그램과 연결해 준다. 또한 가상기계에서 제공하는 API 중에서 기본적인 연산에 관련된 것들은 가상기계 코어의 인터프리터에서 수행하지만 입출력, GUI, 통신 등에 관한 부분은 해당 플랫폼의 API를 이용해서 작업을 수행한다. 따라서 가상기계를 특정 플랫폼에 탑재하기 위해서는 플랫폼의 특성인 자료형, 사용가능한 자원, 소프트웨어적으로 지원하는 함수, 파일 관리 기법 등에 대한 기반 연구가 선행되어야 한다.

※ 이 논문은 2003년도 상지대학교 교내 연구비 지원에 의한 것임.
† 종신회원 : 상지대학교 컴퓨터정보공학부 조교수
논문접수 : 2005년 6월 14일, 심사완료 : 2005년 8월 11일

본 논문에서는 다양한 플랫폼에 가상기계를 효율적으로 탑재할 수 있는 어댑터 모델을 설정한 후 효과적인 네이티브 함수 연결 기법을 제시하였다. 이를 위해, 첫째, 기존의 KVM, WabaVM을 특정 플랫폼에 탑재하는 기법과 네이티브 함수 연결 기법을 고찰한 후 새로운 어댑터 모델을 설계하였다. 둘째, 플랫폼에서 지원하는 네이티브 함수를 효과적으로 호출할 수 있도록 네이티브 함수 테이블(native function table)을 설계하고 구현하였다. 마지막으로 개발된 어댑터를 적용하여 실제로 Linux, Windows 환경에 탑재한 후 수행되는 실험 및 분석 결과를 제시하였다.

본 논문의 구성은 2장에서 가상기계를 실제로 탑재하는 기법과 고려사항에 대해 기술하며 기존의 네이티브 함수 연결 기법에 대해서도 관련 연구 기법을 기술하였다.3장에서는 기존 가상기계의 어댑터 분석 결과를 기반으로 새로운 어댑터 구현 모델을 제시하며 효율적인 네이티브 함수 호출을 위해 설계하고 구현된 네이티브 함수 테이블 구조 및 기능에 대해 기술하였다. 또한 어댑터의 구현상에서 실제적으로 네이티브 함수 호출시에 발생되는 과정 및 동작에 대해서도 기술하였으며 본 연구에서 완성된 어댑터를 실제 운영체제에 탑재하여 그 결과와 실험 분석 결과도 기술하였다. 마지막으로 4장에서는 본 논문의 결론과 향후 연구 방향 및 보완 부분에 대해 기술하였다.

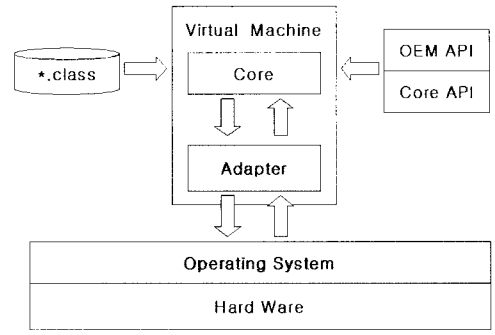
2. 어댑터 및 네이티브 함수 연결

2.1 어댑터

가상기계 구현은 가상기계의 핵심 동작을 구현하는 부분과 특정 플랫폼과 인터페이스 부분을 담당하는 어댑터(adapter) 기능을 구현하는 부분으로 크게 구분된다. 따라서 가상기계의 핵심 부분은 플랫폼에 독립적인 부분으로서 특정 플랫폼에 탑재하더라도 코드의 수정 없이 구현된다. 어댑터는 가상기계가 탑재되는 플랫폼의 특성에 의존하여 구현되는 부분으로서 가상기계의 핵심 부분이 플랫폼에 독립적으로 작동을 할 수 있도록 플랫폼의 API를 이용해서 구현된다. 가상기계를 다른 플랫폼에 탑재할 때마다 새롭게 설계하고 만드는 작업은 상당히 비효율적인 방법이며 가상기계를 개발시에 이식성을 충분히 고려해서 설계 및 제작되어야 한다[3].

일반적으로 가상기계의 개발은 다양한 플랫폼에 원활히 탑재될 수 있도록 (그림 1)과 같이 가상기계를 핵심 부분과 탑재를 위한 부분으로 분리하여 구현한다[3][4].

가상기계 코어(Core) 부분은 자바 언어의 *.class 파일과 같은 실행 파일 포맷을 로딩하는 로더와 실질적인 실행을 담당하는 인터프리터로 구성되어 있다. 가상기계를 플랫폼에 탑재하는 기능을 담당하는 어댑터는 플랫폼과 가상기계 코어간의 인터페이스 역할을 담당한다. 어댑터 구현시에 운영체제와 가상기계간에 실질적인 인터페이스를 담당하는 네이티브 함수 부분이 중요한 역할을 하며 이러한 네이티브 함수는 네이티브 함수 테이블을 통해 소스 프로그램의 가상

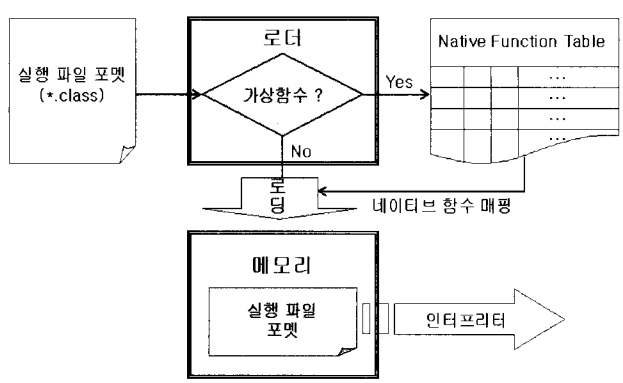


(그림 1) 가상기계 탑재 구조

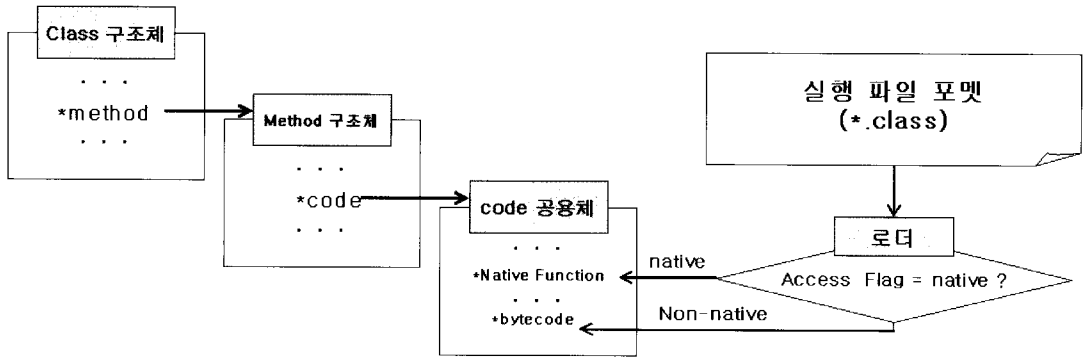
함수와 매핑될 수 있으며 가상함수를 추가하고 네이티브 함수 테이블과 네이티브 함수부를 재구성함으로써 가상기계가 지원하는 기능을 계속해서 확장시킬 수 있다. 또한 가상기계의 속도가 우선적으로 고려된다면 거의 모든 API들을 네이티브 함수로 구현함으로써 크기는 커지지만 상당히 빠른 실행 속도를 얻을 수 있는 장점을 가지고 있다.

가상기계의 로더에서 파일을 메모리에 로딩하는 중에 가상함수를 만나게 되면 (그림 2)와 같이 파일에 있는 코드 대신 네이티브 함수를 저장하고 있는 네이티브 함수의 포인터를 로딩하게 되며 가상함수와 네이티브 함수의 매핑 관계는 네이티브 함수 테이블에 명시 되어있다. 네이티브 함수 테이블을 통해 가상함수와 1:1 매핑하는 함수를 네이티브 함수라 부르며 네이티브 함수에서 실질적인 시스템 호출이 발생하게 된다[7][11].

네이티브 함수는 가상기계 상의 언어에서 가상함수를 대신 하는 가상기계 내부에 있는 함수로서 시스템 함수 호출뿐만 아니라 다른 나머지 함수들도 가상 함수를 통해 네이티브 함수를 호출하는 형태로 구현될 수 있으며 네이티브 함수에 의존도가 높아질수록 가상기계의 속도는 증가하지만 부피는 커지게 된다. 그러므로 반드시 필요한 시스템 호출 함수만 네이티브 함수로 구현하는 것이 보편적이며 이를 통해 가상기계가 특정 플랫폼에 탑재 될 수 있다. 네이티브 함수로 구현해야 할 부분들은 프로그램이 수행되기 위해서 시스템에 종속적일 수밖에 없는 부분으로서 기본적인 입출력, Network, GUI, 디바이스 요청 등과 같은 부분을 수행해야 한다.



(그림 2) 네이티브 함수와 네이티브 함수 테이블



(그림 3) 자바 클래스 구조체 및 네이티브 함수 로딩 과정

2.2 네이티브 함수 실행 기법

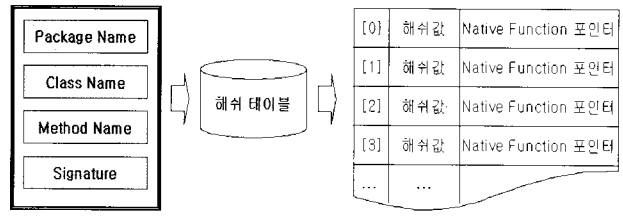
WabaVM은 PalmOS®상에서 작동할 목적으로 개발된 가상기계이다. WabaVM의 특성은 자바 문법을 사용하고 있지만 스레드, 예외처리 등을 지원하지 않고 있으며 최상위 클래스인 Object 클래스에 대해서는 문자열 하나만 가지고 있는 형태로 매우 간결하게 구현 되어 있다. 특히, PalmOS®의 기본 인터페이스가 GUI이기 때문에 GUI 인터페이스를 제공하는 API에 대해서는 많은 클래스들을 제공하고 있다[7].

KVM은 소형기기를 위한 CLDC를 작동시키기 위해 만들어진 가상기계로서 WabaVM에 비해 규모는 상당히 큰 편이다. 특히, WabaVM에 비해 API에 대한 확장성이 뛰어나며 가상기계의 특정 플랫폼 탑재 기법에 있어서 효율적은 방법을 제시하고 있다[10].

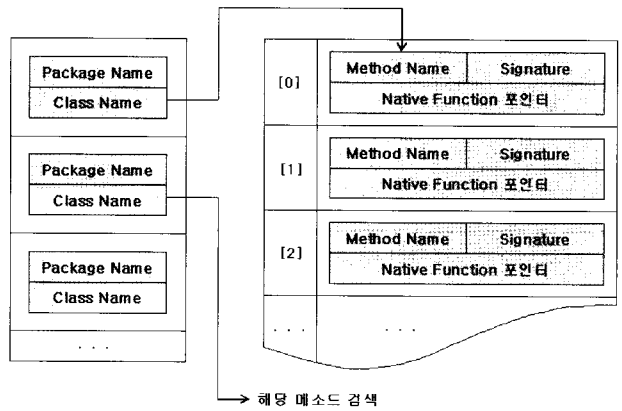
WabaVM[17], KVM[10]에서 네이티브 함수의 로딩은 로더에 의해 수행되며 Java 언어의 경우 native 액세스 플래그를 통해 가상함수와 메소드를 구별한다. 즉, native 액세스 플래그는 가상기계 내부적으로 로더의 로딩 과정과 인터프리터의 실행 과정에서 가상함수의 구분이 필요 있을 때 사용되는 플래그이다. 네이티브 함수의 로딩 방식은 WabaVM, KVM, JVM 모두가 동일한 방식으로 이루어지고 있으며 네이티브 함수 로딩에 관한 방식은 로더가 클래스 파일을 저장할 수 있는 (그림 3)과 같은 class 구조체에 중간 코드를 로딩하는 과정에서 이루어지고 있다.

2.3 네이티브 함수 테이블 구조

네이티브 함수 테이블은 API 상에 있는 가상함수가 어떤 네이티브 함수와 매핑을 이루는지를 명시하고 있는 테이블로서 로더에서 사용되는 테이블이다. 테이블을 구성하고 있는 요소는 가상함수를 구별하기 위한 Package name, Class name, Method name, Signature와네이티브 함수를 가리키는 포인터로 구성되어 있다. WabaVM의 네이티브 함수 테이블은 메모리가 부족한 PalmOS®에 적합하게 아주 적은 양의 메모리를 점유하도록 설계되어 있다. 가상함수를 파악하기 위한 정보(Package Name, Class Name, Method Name, Signature)들을 전부 저장하는 것이 아니고 이 자료들을 가지고 해시 값을 구한 다음에 해시 값을 해시 테이블의 인덱스로 사용하는 것이 아니라 배열에 해시 값을 넣어 배열을



(그림 4) WabaVM의 네이티브 함수 테이블 구조



(그림 5) KVM의 네이티브 함수 테이블 구조

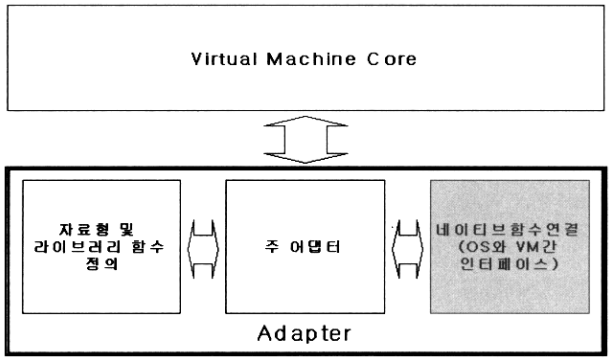
검색 하는 방식을 사용하고 있다. 검색이 이루어지는 동한 매번 해시 값을 구하는 작업 및 모든 가상함수의 리스트가 열거되어 있어서 시간적으로는 테이블의 효율이 떨어지는 단점이 있다. 또한 테이블에 함수를 추가하기 위해서는 필요한 정보들을 모두 모아서 계산을 해서 테이블을 재구성해야 한다는 단점을 가지고 있다[12][14].

KVM의 네이티브 함수 테이블은 클래스의 이름을 가지고 있는 배열이 있으며 각각의 구성 요소는 자신의 가상함수 배열을 링크하고 있는 구조로 이루어져 있다. 모든 정보가 테이블에 저장되어 있으므로 테이블의 크기는 큰 편이라고 할 수 있으나 검색 속도는 WabaVM과 같은 계산과정 없이 클래스 이름으로 먼저 검색을 하고 메소드 이름으로 검색을 하기 때문에 검색 속도는 WabaVM에 비해서 상대적으로 빠르다. 또한 API 확장으로 인하여 네이티브 함수를 추가해야 할 경우에도 단순히 명칭을 가지고 구성 할 수 있으므로 테이블의 확장 및 재구성이 용이하다.

3. 어댑터 및 네이티브 함수 테이블

3.1 어댑터 구현 모델

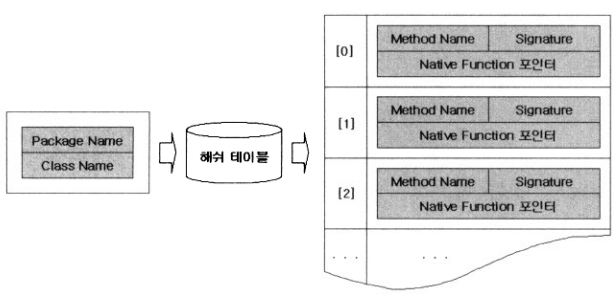
어댑터 구조는 기존의 WabaVM, KVM에 기반을 두고 있으며 가상기계 코어의 플랫폼 독립성을 보존하기 위해 전체 가상기계에서 플랫폼에 의존적인 부분을 분리한 후 어댑터 구조를 크게 3부분으로 나누어 구현하였다. 어댑터의 구성은 자료형의 크기를 맞추어 주는 정의 부분, 가상기계 코어의 입출력과 실행을 위한 인터페이스를 제공하는 주 어댑터 부분, 마지막으로 운영체제와 가상기계간에 실질적인 인터페이스를 담당하는 네이티브 함수 부분으로 (그림 6)과 같이 구성되어 있다. 특히 본 논문에서는 어댑터의 구현에서 가장 중요한 역할을 담당하는 네이티브 함수 연결 부분에서 보다 효율적인 네이티브 함수 연결을 위해 네이티브 함수 테이블을 새롭게 설계하고 구현하였다.



(그림 6) 어댑터 구현 모델

3.2 네이티브 함수 테이블의 설계 및 구현

WabaVM에서는 새로운 네이티브 함수의 확장 및 추가가 어렵다는 단점이 있으나 메모리를 적게 사용하는 장점을 가지고 있다. 이에 비해 KVM은 네이티브 함수의 확장 및 추가에 있어서는 쉬운 인터페이스를 제공하나 크기가 크다는 단점이 지적되어 왔다. 본 논문에서는 이러한 요소를 적절하게 완충하여 거의 유동적이지 않은 패키지, 클래스 이름에 대해서는 해쉬 값을 이용하여 메모리를 절약하고 추가 삭제가 많은 가상함수에 대해서는 가상함수 이름을 문자열 정보로 그대로 사용하도록 하는 네이티브 함수 테이블을 (그림 7)과 같이 설계하였다.



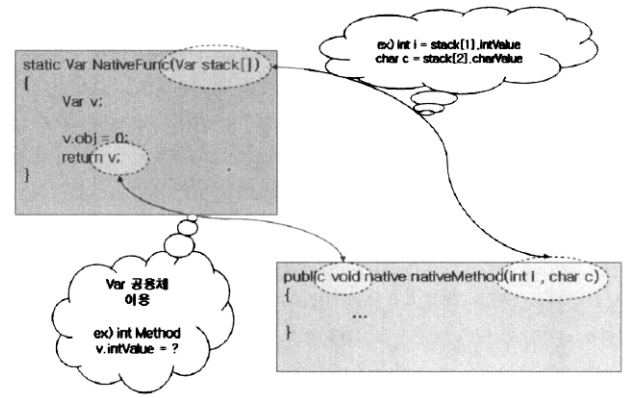
(그림 7) 네이티브 함수 테이블 설계

실질적인 수행을 위한 네이티브 함수 인터페이스는 실행 스택(execute stack)에 직접 접근하는 방식을 이용하면 실행 스택에 대한 정보를 잘못 연산을 수행할 수 있기 때문에 실행 스택으로부터 해당 가상함수의 매개변수가 처음으로 로딩되어 있는 시작 포인터를 넘겨받고 해당 연산을 실행한 후 실행 스택에 반환하는 구조로 구현하였다.

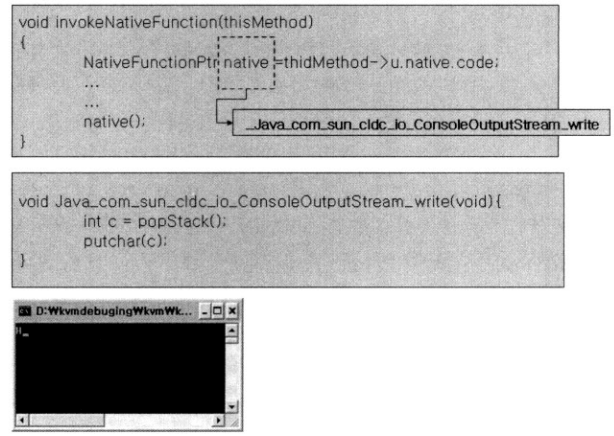
3.3 네이티브 함수 실행

네이티브 함수의 실행은 인터프리터에서 실행 파일 포맷에 대한 실행중에 native 네이티브 억세스 플래그를 만나게 되면 code 공용체의 네이티브 함수 포인터를 이용하여 해당 함수를 호출하게 되며 호출을 받은 네이티브 함수는 인터프리터의 실행 스택에 접근하여 가상 함수의 매개변수, 반환 값, 자신이 속한 객체의 필드 값에 접근하여 실행을 한다.

네이티브 함수의 인터페이스는 (그림 8)과 같이 모든 자료형을 저장할 수 있는 var 공용체를 통해 가상함수의 매개변수 및 반환 값을 처리한다. 가상함수의 매개변수에 관한 데이터는 인터프리터의 실행 스택에서 해당 시작 주소를 stack[]에 저장해 전달함으로써 배열에서 사용하는 것처럼 사용할 수 있게 되며 모든 작업을 마치고 나면 기존의 가상함수의 반환 값에 적합한 자료형을 var형 변수에 저장하여 반환한다.



(그림 8) 네이티브 함수의 인터페이스



(그림 9) 네이티브 함수의 실행 예

비정적(non-static) 가상함수는 가상함수가 객체 멤버로써 동작하기 때문이며 실행 스택에 객체에 대한 정보가 저장되어 있어야 한다. 정적 가상함수의 경우는 클래스 내부에 포함 되어 있지만 별도의 메모리 공간을 차지하고 있는 함수로서 객체에 종속적이지 않고 함수 자체적으로 기능을 수행할 수 있는 함수이다. 이러한 정적 가상함수는 네이티브 함수의 구현에 있어서 객체에 대한 정보가 필요하지 않으며 매개 변수만 받아 연산을 수행한 후 결과를 반환 하게 된다.

네이티브 함수가 객체로부터 필드 값을 받아오고 변경하는 등의 작업을 하기 위해서는 Object에 필드 값을 접근할 수 있는 기법이 제공되어야 하는데 필드에 대한 정보는 로더에서 클래스 정보를 읽어 들여 저장하는 class 구조체의 멤버에 저장한다. 네이티브 함수가 class 구조체로부터 필드 값을 가져오는 방식은 인터프리터로부터 Object에 대한 식별 번호를 얻어와서 이 번호를 통해서 해당 객체에 접근하다.

가상함수가 네이티브 함수를 통해서 실행되는 과정을 검증하기 위해 (그림 9)와 같은println() 메소드는 가상 함수인 write() 메소드에 한 문자씩 전달하며 호출하는 구조로 이루어져있다. 이러한 write() 메소드는 표준 출력을 수행하는 가상함수이기 때문에 C언어 함수인 putchar()로 구현된 네이티브 함수를 호출함으로써 화면상에 문자를 출력한다.

3.4 실험 결과 및 분석

본 연구에서는 기존의 WabaVM의 코어 부분을 변경 없이 사용하고 어댑터 부분을 본 논문에서 제시한 설계 모델에 따라 구현하였다. 특히, 가상기계의 어댑터 구현에서는 보다 효율적인 네이티브 함수 연결을 위해 새로운 네이티브 함수 연결 테이블의 구조를 정의한 후 이를 적용하였다. 또한 실제적으로 최종 완성된 가상기계를 Liunx 및 Windows 환경<표 1>에 탑재하여 수행한 실험 결과는 (그림 11), (그림 12)와 같다.

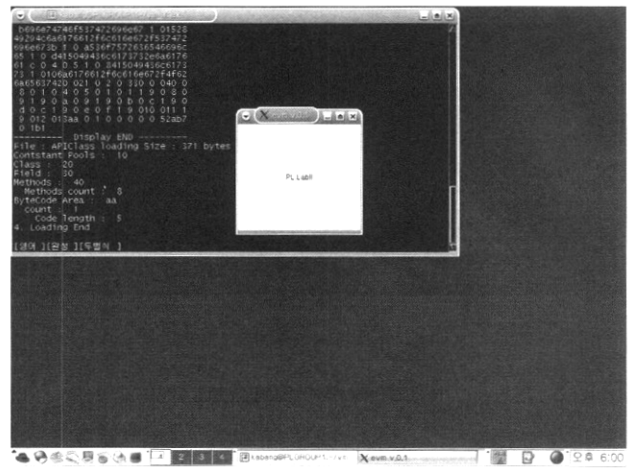
〈표 1〉 실험환경

	LINUX	WINDOWS
하드웨어	1.3Ghz , 256MB	1Ghz , 512MB
운영체제	Red Hat Linux 9 (Kernel - 2.4.20)	MS Windows XP Pro SP2
컴파일	GCC 3.2.2	Microsoft Visual C++ 6.0

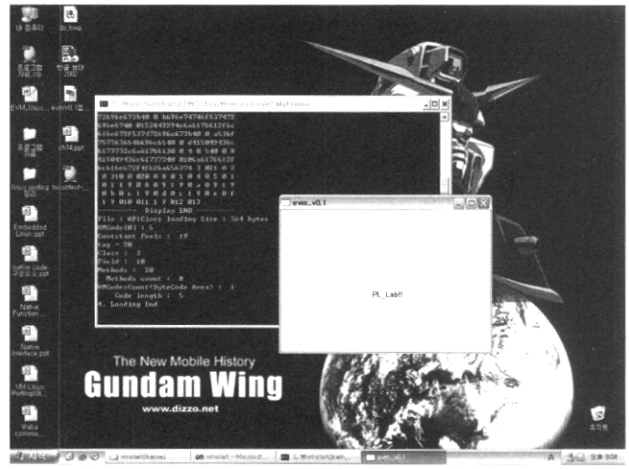
MyFrame.java

```
public class MyFrame {
        public static void main(String[] args) {
                APIClass.outFrame(10);
        }
}
```

(그림 10) 프레임 생성을 위한 예제 코드



(그림 11) Linux 환경에서 실행 결과



(그림 12) Windows 환경에서 실행 결과

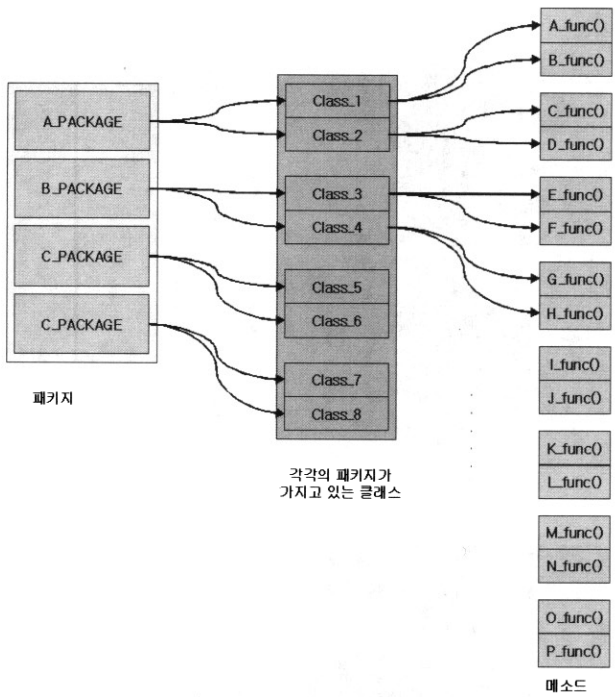
〈표 2〉 네이티브 함수 테이블 크기 비교

	네이티브 함수 테이블 크기 계산
WabaVM	(int + 함수포인터) * 메소드수
KVM	((스트링 + 스트링 + 포인터) * 클래스수) + ((스트링 + 스트링 + 함수포인터) * 메소드수)
New VM	((int + 포인터) * 총 클래스수) + ((스트링 + 스트링 + 함수포인터) * 총 메소드수)

〈표 3〉 네이티브 함수 테이블 확장에 필요한 작업

	네이티브 함수 테이블 크기 계산
WabaVM	메소드가 속한 패키지, 클래스, 매소드, 시그니쳐 스트링 모두에 대한 해쉬 값을 계산해서 테이블에 추가
KVM	메소드, 시그니쳐 스트링, 함수포인터 추가
New VM	메소드, 시그니쳐 스트링, 함수포인터 추가

가상기계 탑재 실험은 Linux, Windows 환경에서 수행하였으며 표준 출력과 각각의 플랫폼에 맞는 GUI 테스트를 위해 (그림 10)과 같은 프레임 실행에 관한 테스트용 API에 맞는 네이티브 함수를 작성하여 실험하였다.



(그림 13) 네이티브 함수 검색 속도 측정을 위한 예제(패키지, 클래스, 메소드)

실제로 본 연구에서 설계한 네이티브 함수 테이블을 적용하여 구현된 가상기계(New VM)와 WabaVM, KVM에서 사용한 네이티브 함수 테이블 크기를 계산하는 과정은 <표 2>와 같다. New VM에서 사용한 네이티브 함수 테이블의 크기는 KVM에서 사용한 테이블의 크기보다 월등히 작지만 WabaVM에서 사용한 네이티브 함수 테이블의 크기보다는 약간 큰 결과를 제시하고 있다. 하지만 클래스의 수가 작은 경우에는 WabaVM의 크기와 거의 동일하다는 것을 실험 결과를 얻었다.

기존 네이티브 함수 테이블에 새로운 메소드를 추가하여 확장할 경우 부득히하게 테이블의 수정 작업을 거치게 된다. 따라서 네이티브 함수 테이블의 구조에 따라 <표 3>과 같은 오버헤드가 발생하게 된다.

기존 네이티브 함수 테이블에 새로운 메소드를 추가할 경우 WabaVM은 전체적인 해쉬 값을 다시 계산해야 하는 부담을 가지고 있지만 KVM과 본 논문에서 제시한 네이티브 함수 테이블은 상대적으로 적은 부담을 가지고 있다. 즉, WabaVM은 메소드 추가에 대해 해쉬 값을 다시 계산해야 하는 부담을 갖지만 KVM은 명칭이 스트링 형태로 구성되어 있어 확장이 매우 용이하지만 크기가 방대해지는 단점을 가지고 있다. 본 논문에서 제시한 기법은 클래스를 추가할 경우에만 해쉬 값을 다시 계산하고 메소드 추가시에는 KVM과 동일한 방식으로 수행되고 있다. 따라서 확장성에는 WabaVM, KVM에 비해 상대적으로 장점을 가지고 있다.

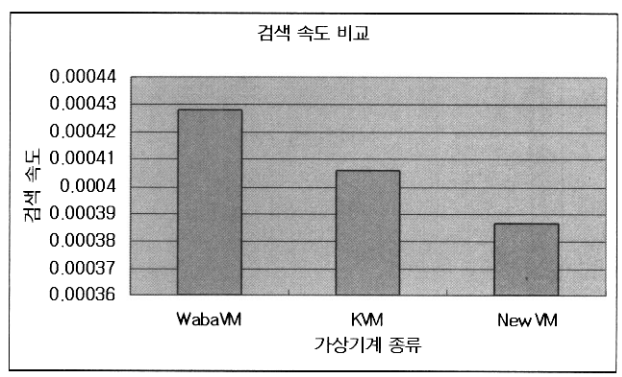
실제적으로 네이티브 함수 테이블을 통해 검색되는 네이티브 메소드의 속도를 측정하기 위해 (그림 13)과 같은 구조를 가진 예제 프로그램을 구성한 후 WabaVM, KVM 및 본 논문에서 제안한 네이티브 함수 테이블을 이용하여 속도를 측정하였다.

(그림 13)에 대해 WabaVM, KVM과 본 연구에서 새롭게 구현한 네이티브 함수 테이블을 이용한 네이티브 함수 검색 속도 결과는 <표 4>와 같다.

위 검색 속도 측정 결과를 통해 WabaVM은 검색하고자 하는 네이티브 함수를 발견할 때까지 해쉬 값에 의한 숫자 비교를 통해 전체 검색이 이루어지는 것을 확인할 수 있으며 KVM은 먼저 일치하는 클래스를 검색한 후 클래스에 대한 네이티브 함수 검색을 수행하므로서 검색 속도가 감소함을 확인할 수 있다. 마지막으로 새롭게 설계된 네이티브 함수 테이블을 이용한 경우에는 클래스 이름에 대한 검사는 해쉬 값을 통한 숫자를 비교한 후 메소드 검색에는 좀 더 빠른 검색이 발생함을 확인할 수 있다. 전체적인 검색 속도의 비교는 (그림 14)와 같이 새롭게 설계된 네이티브 함수 테이블을 이용하는 경우가 검색 속도에서 WabaVM 및 KVM에 비교하여 효율성이 있는 것을 알 수 있다.

〈표 4〉 네이티브 함수 검색 속도 측정

패키지	클래스	네이티브함수	WabaVM	KVM	New VM
A_PACKAGE	Class_1	A_func	0.000403	0.000402	0.000400
		B_func	0.000401	0.000395	0.000391
	Class_2	C_func	0.000438	0.000436	0.000403
		D_func	0.000418	0.000413	0.000395
B_PACKAGE	Class_3	E_func	0.000431	0.000423	0.000396
		F_func	0.000402	0.000379	0.000362
	Class_4	G_func	0.000391	0.000379	0.000349
		H_func	0.000435	0.000428	0.000380
C_PACKAGE	Class_5	I_func	0.000432	0.000418	0.000369
		J_func	0.000419	0.000403	0.000392
	Class_6	K_func	0.000436	0.000383	0.000365
		L_func	0.000379	0.000362	0.000361
D_PACKAGE	Class_7	M_func	0.000567	0.000415	0.000403
		N_func	0.000438	0.000408	0.000398
	Class_8	O_func	0.000428	0.000422	0.000415
		P_func	0.000427	0.000425	0.000405



(그림 14) 네이티브 함수 검색 속도 비교

4. 결론 및 향후 연구 방향

가상기계에서 수행되는 응용 프로그램은 플랫폼 독립적인 특성을 갖지만 가상기계의 다양한 요소에 대해서는 의존적인 성질을 갖는다. 즉, 플랫폼 환경에서 작동하고 있는 가상기계라는 응용 프로그램의 한계를 가지고 있다. 가상기계를 특정 플랫폼에 탑재하는 것은 가상기계라는 프로그램이 특정 플랫폼의 자원을 충분히 활용할 수 있도록 메모리 크기, 자료형, 입/출력 장치의 인터페이스 등을 고려하여 가상기계를 재구성하는 것이다.

어댑터는 가상기계를 이용하여 응용 프로그램을 원활히 수행할 수 있도록 플랫폼의 특성에 의존하는 입출력 함수, 시스템 호출 함수 등과 같은 플랫폼에 의존적인 네이티브 함수를 적절한 방식으로 응용 프로그램과 연결해 준다. 따라서 가상기계를 특정 플랫폼에 탑재하기 위해서는 플랫폼의 특성인 자료형, 사용가능한 자원, 소프트웨어적으로 지원하는 함수, 파일 관리 기법 등에 대한 기반 연구가 선행되어야 한다.

본 논문에서는 다양한 플랫폼에 가상기계를 효율적으로 탑재할 수 있는 어댑터 모델을 설정한 후 효과적인 네이티브 함수 연결 기법을 제시하였다. 이를 위해, 첫째, 가상기계를 다양한 플랫폼에 탑재할수 있는 새로운 어댑터 모델을 설계하였다. 둘째, 네이티브 함수를 효과적으로 호출할 수 있도록 네이티브 함수 테이블을 설계하고 구현하였으며 개발된 어댑터를 적용하여 실제로 Linux, Windows 환경에 탑재한 후 수행되는 실험 및 분석 결과를 제시하였다.

본 논문에서 설계하고 구현한 네이티브 함수 테이블의 크기는 KVM에서 사용한 테이블의 크기보다 작지만 WabaVM에서 사용한 네이티브 함수 테이블의 크기보다는 큰 결과를 얻었으며 네이티브 함수 테이블에 새로운 메소드를 추가할 경우 WabaVM은 전체적인 해쉬 값을 다시 계산해야 하는 부담을 가지고 있지만 KVM과 본 논문에서 제시한 네이티브 함수 테이블은 상대적으로 적은 부담을 가지고 있다. 마지막으로 네이티브 함수 검색 속도면에서는 WabaVM은 검색하고자 하는 네이티브 함수를 발견할 때까지 해쉬 값에 의한 숫자 비교를 통해 전체 검색이 이루어지므로 검색 속도면에서 KVM과 본 논문에서 제시한 기법에 비해 느린 속도로 진행된다. 또한 KVM은 먼저 일치하는 클래스를 검색한 후 클래스에 대한 네이티브 함수 검색을 수행하므로서 검색 속도가 감소함을 확인할 수 있다. 결론적으로 본 논문에서 제시한 네이티브 함수 검색 속도는 WabaVM 및 KVM에 비교하여 효율성을 갖는다.

향후 보다 다양한 실험 결과를 제시하기 위해서는 네이티브 함수를 보다 많이 추가하여 네이티브 함수 테이블 크기, 네이티브 함수 검색 속도에 대한 보완 연구가 진행되어야 한다. 또한 클래스에 속하는 메소드 개수와 클래스 명칭 및 메소드 명칭의 길이에 따라 네이티브 함수 검색 속도가 달라지는 경우에 대한 보완 연구가 진행되어야 한다. 마지막으로 가상기계를 다양한 플랫폼에 자동적으로 탑재하기 위한 재목적 기법에 대한 연구가 진행되어야 한다.

참 고 문 헌

[1] Jon Meyer & Troy Downing, "Java Virtual Machine," March, 1997.

[2] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification," 2nd edition, Addision-Wesley, 1999.

[3] Bill Venners, "Inside the Java Virtual Machine," McGraw-Hill, 1998.

[4] Bill Blunden, "Virtual Machine Design and Implementation in C/C++," Wordware Publishing, Inc., 2002.

[5] Joshua Engel, "Programming for the Java Virtual Machine," Addison-Wesley, 2000.

[6] Rainer Leupers and Peter Marwedel, "Retargetable Compiler Technology for Embedded System: Tools and Applications," Kluwer Academic Publishers, 2001.

[7] John Whaley, "Joeq: A Virtual Machine and Compiler Infrastructure," In the Proceddings of ACM SIGPLAN Conferences on Interpreters, Virtual Machines and Emulators 2003(IVME '03), ACM Press, pp.58-66, 2003.

[8] Wen-mei W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," The Proceeding of the 29th Annual International Symposium on Microarchitecture, Dec., 1996.

[9] Mahadevan Ganapathi, Charles N.Fisher and John L. Hennessy, "Retargetable Compiler Code Generation," ACM Computing Surveys, Vol.14, No.4, 1982.

[10] Sun Microsystems, "The K Virtual Machine(KVM) White Paper. Technical report," Sun Microsystems, 1999.

[11] John R. Levine, "Linkers and Loaders," Morgan Kaufmann Publishers, 2000.

[12] 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.

[13] W. Paugh, "Compressing Java class files," In the Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation(PLDI) '99, pp.247-258, May, 1999.

[14] Clausen, L.R., Schultz, U.P., Consel, C., Muller, G., "Java Bytecode Compression for Low-End Embedded Systems," ACM TOPLAS, Vol.22, No.3, pp.471-489, May, 2000.

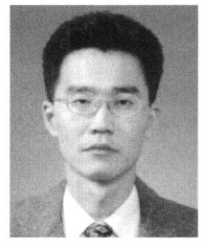
[15] Derek Rayside, Evan Mamas, Erik Hons, "Compact Java Binaries for Embedded System," Proceddings of the 9th NRC/IBM centre for Advanced Studies Confercence(CASCON '99), pp.1-14, 1999.

[16] Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D., "Practical Extraction Techniques for Java," ACM TOPLAS, Vol.24, No.5, pp.625-666, Nov., 2002.

[17] Waba Programming Platform,http://www.wabasoft.com

[18] Joeq Virtual Machine, http://sourceforge.net/projects/jeoq

[19] Jike Research Virtual Machine, http://www.ibm.com/developerworks/oss/jikesrvm



고 광 만

e-mail : kkman@mail.sangji.ac.kr

1991년 원광대학교 컴퓨터공학과(공학사)

1993년 동국대학교 컴퓨터공학과 (공학석사)

1998년 동국대학교 컴퓨터공학과 (공학박사)

1998년~2001년 광주여자대학교 컴퓨터학부 전임강사

2002년~2003년 Queensland University of Technology 연구교수

2001년~현재 상지대학교 컴퓨터정보공학부 조교수

관심분야 : 프로그래밍 언어 설계 및 구현