論文
5-9-14

임베디드 시스템을 위한 가상기계의 라이브러리 링킹에 관한 연구

A Study on the Library Linking of a Virtual Machine for Embedded System

고광만(Ko Kwang Man)[1]

요 약

본 논문은 소규모 장치 및 모바일 장치 등에 탑재되고 있는 KVM, Waba VM의 탑재 기법 및 네이티브 코드 연결 기법을 기반으로 임베디드 시스템에 적합한 가상기계의 정적/동적 라이브러리 연결 기법에 관한 연구이다. 이를 위해, KVM, Waba VM의 네이티브 함수 연결 기법을 기반으로 정적/동적 라이브러리 함수 연결을 위한 새로운 네이티브 함수 테이블을 구현하였다. 또한 구현된 기법을 이용하여 다양한 실험 및 분석 결과를 제시하였다.

키워드 : 가상기계, 정적/동적 라이브러리, 로딩/링킹, 임베디드 시스템

Abstract

This paper presents the experiences of the static and dynamic library function connection technique for the embedded virtual machines, base on the native function connection methods of the virtual machines such as KVM, Waba VM. For this goals, we implements the new native function table for the static and dynamic library function connection technique base on the native function connection methods of the virtual machines such as KVM, Waba VM. And we presents the variety experiment and analysis results using the implemented technique.

Keywords : Virtual Machine, Static/Dynamic Library, Loading/Linking, Embedded System

논문접수 : 2004. 11. 15.
심사완료 : 2004. 12. 3.

1) 정회원 : 상지대학교 컴퓨터정보공학부 조교수

❖ 논문은 2002년도 상지대학교 교내 연구비 지원에 의한 것임

1. 서 론

임베디드 시스템을 위한 가상기계의 설계 및 개발 기술 분야에 대한 국내 연구는 모바일 디바이스를 위한 플랫폼 환경에 관한 연구가 활발히 진행되고 있지만 PDA, 디지털 TV 분야 등에서 가상기계 환경을 제공하기 위한 시도는 미약하며 마이크로소프트사의 C#, SUN사의 자바 언어 등을 모두 수용할 수 있는 통합 플랫폼 또는 모바일 분야를 포함한 모든 임베디드 시스템을 위한 플랫폼의 개발은 이미 시장이 형성된 휴대폰, PDA, 그리고 앞으로 형성될 대규모 디지털 TV와 셋톱 박스(set-top box) 등을 수용하기에 매우 부족하다. 또한, 모바일 디바이스 환경에 적합한 컴파일러 기술에 대한 연구도 프로그래밍 언어나 플랫폼에 의존적이어서 프로그래머가 프로그램을 한번 작성하면 프로세서나 운영체제와 같은 플랫폼에 의존하지 않고 어느 시스템에서나 실행할 수 있는 기술에 대한 연구는 매우 미흡한 실정이다.

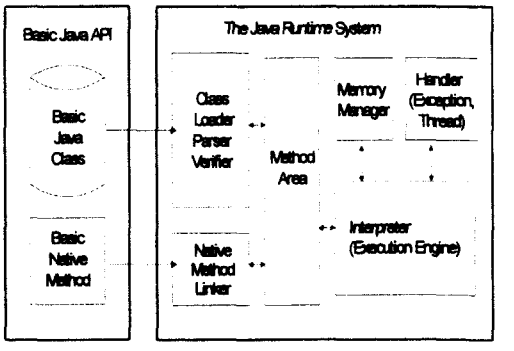
가상기계(virtual machine)란 프로세서, 운영체제 등이 바뀌더라도 응용 프로그램을 변경하지 않고 사용할 수 있는 기술로 특히, 임베디드 시스템을 위한 가상기계 기술은 모바일 디바이스와 디지털 TV 등에 탑재할 수 있는 핵심기술로 다운로드 솔루션에서는 꼭 필요한 소프트웨어 기술이다. 응용 프로그램의 개발 방법 및 실행 방법은 크게 네이티브 애플리케이션과 가상기계 애플리케이션으로 나눌 수 있으며 전자는 이제까지 사용하던 방법으로 실행속도 면에서는 탁월한 장점을 갖는다. 그러나 플랫폼이 변경되면 모든 응용프로그램을 변경해야 할 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[5][7].

본 연구에서는 임베디드 시스템에서 위한 가상기계의 구축 시에 필요한 최적의 실행 환경을 지원하기 위해 반드시 필요한 정적 및 동적 라이브러리 연결에 기법을 연구한다. 특히, 동적 라이브러리 연결에 관한 연구는 보다 효율적인 응용 프로그램의 실행을 위해 최근 사용되는 고급 언어에서 반드시 요청되는 기술로서 이를 지원하기 위한 국내외 연구가 활발히 진행되고 있다.

2. 관련 연구

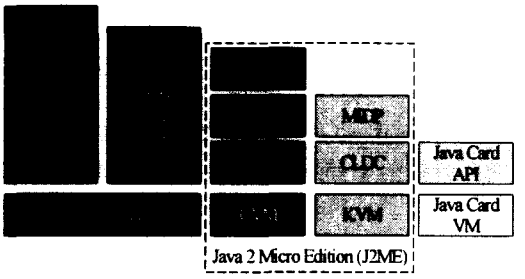
2.1 가상 기계 종류 및 구조

JVM(Java Virtual Machine)은 자바 프로그램을 실질적으로 실행하는 엔진으로서 231개의 바이트코드 명령어 집합과 [그림 1]과 같이 클래스 로더 및 검증기, 메모리 관리자, 에러 및 예외 처리기, 네이티브 메소드 링커, 인터프리터 등으로 구성되어 있다. 가상기계를 사용함으로써 이식성과 보안에 장점을 갖지만 가상기계를 실질적인 목적기계로의 이식, 컴파일 방식의 언어에 비해 실행속도가 느린 단점을 가지고 있다. JVM을 임베디드 시스템에 적용하기 위해서는 특정한 디바이스에 적합한 환경을 지원하기 위해 적은 메모리 사용 등과 같은 특정한 제약 사항과 특정한 디바이스를 위한 기능 및 이를 지원할 수 있는 API의 지원이 요구된다[5][7].



[그림 1] 자바 가상 기계의 구조

J2ME 플랫폼에 적합한 KVM(Kilobyte Virtual Machine)은 J2SE, J2EE와 달리 다중 사양을 수용하고 있으며 특히, 메모리 요구 사항에 따라 [그림 2]와 같이 CLDC와 CDC로 구분된다[7].

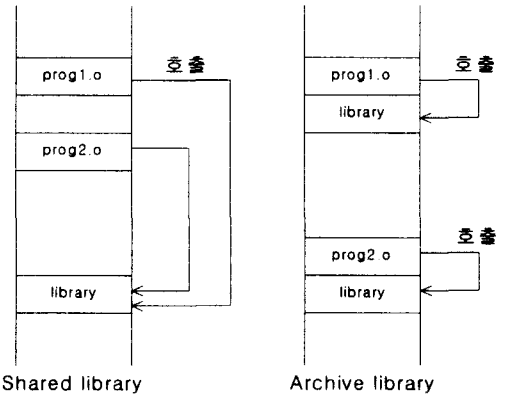


[그림 2] Java2 플랫폼 : KVM의 구조

CLDC는 KVM을 가상기계로 채택하고 있다. 이러한 KVM은 작고 한정된 기능을 가진 디바이스에 적합한 컴팩트한 JVM을 구축하기 위해 팜 시리즈를 목표로 한 Spotless 시스템(모바일 디바이스에 적합한 작고 완벽한 자바 가상기계 구현을 목표)이라 불리는 프로젝트로부터 시작되었다. 즉, JVM의 핵심 기술을 유지하면서 단지 수백 킬로바이트의 메모리를 가진 한정된 성능의 디바이스에서도 동작하는 가능한 작고 완벽한 가상기계를 개발하기 위해 시작되었다. 따라서 KVM은 일반적으로 16~32 비트 프로세서, 128~512 KB 메모리, 저전력을 소모하는 휴대폰, 페이저, PDA 등에 탑재된다. 하지만 KVM은 JVM에 비해 JNI, 리플렉션, RMI, 객체 직렬화 등을 지원하지 않는 단점을 가지고 있다.

2.2 정적 및 동적 연결 라이브러리 연결 기법

라이브러리의 종류에는 크게 압축 라이브러리(archive library)와 공유 라이브러리(shared library)로 분류될 수 있다. 압축 라이브러리는 하나 이상의 함수로 구성된 목적 코드들의 집단으로 링커에 의한 연결 과정의 수행 시 정적 바인딩을 이루는 라이브러리이다. 따라서 압축 라이브러리 안에서 필요한 함수는 실행 파일(a.out)내로 복사가 이루어진다. 압축 라이브러리와 마찬가지로 공유 라이브러리도 재배치될 수 있는 목적 코드로 되어 있다. 그러나 링킹 과정에서 공유 라이브러리는 압축 라이브러리와 다르게 취급한다. 링킹 과정이 진행될 때 ld는 필요한 루틴을 a.out으로 복사하지 않고, 그 대신 공유 라이브러리에서 필요한 부분에 대한 정보만을 a.out에 기록하게 된다. 이런 작업을 행하여 주는 링커를 동적 링커(dynamic linker)라 부른다. 실행 파일이 실행될 때, 동적 로더(dynamic loader)는 실행 화일에서 공유 라이브러리가 사용되었는가를 살펴본다. 만약 공유 라이브러리가 사용되었다면, 동적 로더는 기억 장소에 공유 라이브러리를 적재하게 된다. 만약 이미 다른 프로그램에 의해 해당 공유 라이브러리가 메모리에 적재되어 있었다면 재적재(re-loading) 없이 이를 사용한다. 따라서 압축 라이브러리에 비해 코드의 크기가 줄며, 기억 장소를 적게 사용하게 된다는 장점을 갖게 된다. 그러나 공유 라이브러리는 실행 시간 링커에 의해 실행 시간에 실제 바인딩이 이루어지므로 실행 시간 부담을 갖게 된다. 압축 라이브러리와 공유 라이브러리의 메모리 배치 예는 [그림 3]과 같다[2].



[그림 3] 공유 라이브러리와 압축 라이브러리

C++언어의 가상 함수(virtual function) 또는 Java 언어의 오버라이딩(overriding)의 경우에는 컴파일 시간에 객체 또는 식의 자료형을 결정할 수 없는 특성을 가지므로 실행 시간에 실행 정보를 결정해야 하는 동적 바인딩을 지원해야 한다. 가상 함수는 베이스 클래스와 유도된 클래스 모두에서 나타날 수 있다. 이 때 적절한 함수의 호출은 컴파일 시간에 결정할 수 없다. 이와 같은 가상 멤버 함수로의 호출은 컴파일 시간이 아닌 실행 시간에 바인딩이 일어나야 한다. 이를 지원하기 위한 실행 시간 환경을 필요로 한다. 가상 함수로의 호출은 컴파일러에 의해 우선 가상 함수로의 포인터 테이블(Virtual Function Pointer Table; VFPT)을 구성하게 되며 이러한 VFPT은 실행 시간에 의해 검색되며 간접적으로 가상 함수를 호출한다.

2.3 네이티브 함수 연결 기법

네이티브 코드는 가상기계에서 사용할 언어가 제공할 API와 밀접한 관계를 가지고 있다. 이 API는 프로그램이 실행하기 위해서 필수적인 사항으로서 표준 입출력 같은 플랫폼에 종속적인 기능을 사용하기 위해서는 해당 API가 가상 함수로 제작되어 시스템의 해당 기능을 호출하는 구조로 구현되어야 한다. 시스템의 해당 기능의 호출은 가상기계의 인터프리터가 수행중에 내부 구현이 없는 가상 함수를 만나게 되면 가상기계 내부에 있는 네이티브 코드 중에서 해당 기능을 수행할 수 있는 네이티브 코드를 호출하여 인터프리터와는 별개로 실행하여 결과만 돌려주는 형식으로 작동하게 된다. 네이티브 코드부는 가상기계를 구현할 때 사용한 언어를 이용하여 구현하는 것이 보편적이며 가상함수들과는 1:1로 매핑되며 해당 기능들을 수행할 수 있는 여러 개의 함수들로 이루어져 있다. 가상기계상의 언어가 가상기계에서 작동하면서 시스템 호출 등 플랫폼에 종속적인 작업을 수행하기 위해서는 네이티브 코드는 필수적이라 할 수 있으며 가상기계를 특정 플랫폼에 탑재하기 위해서는 해당 플랫폼에 맞추어서 네이티브 코드를 작성해주어야 한다[8].

가상기계의 로더에서 파일을 로딩하여 메모리에 적재하는 중에 가상함수를 만나게 되면 그 부분에 대해서는 파일에 있는 코드 대신 네이티브 코드를 담고 있는 네이티브 함수의 포인터를 대신 적재하게 되며 가상 함수와 네이티브 함수의 매핑 관계는 네이티브 함수 테이블(native function table)에 명시되어 있다. 네이티브 함수 테이블을 통해 가상함수와 1:1 매핑하는 함수를 네이티브 함수라 부르며 실질적인 시스템 호출이 발생되며 가상기계의 네이티브 코드 실행을 위한 부분은 네이티브 함수 테이블과 네이티브 함수 부분으로 나누어진다. 네이티브 함수는 가상기계에서 가상함수를 대신하는 가상기계 내부에 존재하는 함수로서 시스템 호출뿐만 아니라 다른 함수들도 가상함수를 통해 네이티브 함수를 호출하는 형태로 구현될 수 가있으며 네이티브 함수에 대한 의존도가 높아질수록 가상기계의 속도는 증가하지만 부피는 커지게 된다. 따라서 반드시 필요한 시스템 호출 함수만 네이티브 함수로 구현하는 것이 보편적이며 이를 통해 가상기계가 특정 플랫폼에 탑재될 수 있는 것이다. 네이티브 코드로 구현해야 할 부분은 프로그램이 정상적인 수행이 진행되기 위해 시스템에 종속적인 부분으로서 기본적인 입출력, 네트워크, GUI, 디바이스 서비스 요청 등과 같은 부분을 수행해야 하는 부분이다. 이러한 기능을 지원하는 API들은 네이티브 함수를 호출하는 가상함수로 구현되어 있다.

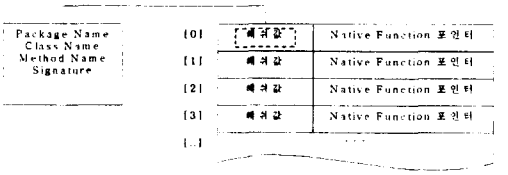
3. 정적/동적 라이브러리 링킹

3.1 네이티브 함수 테이블 구조

네이티브 함수 테이블은 API상에 있는 가상함수가 어떤 네이티브 함수와 매핑을 이루는지

를 명시하고 있는 테이블로서 로더에서 중간 코드를 로딩때 사용된다. 테이블을 구성하고 있는 요소는 가상 함수를 구별하기 위한 패키지 이름, 클래스 이름, 시그네춰, 메소드 이름과 네이티브 함수를 가리키는 함수 포인터로 구성되어 있다. 구현 방식은 JVM, KVM에서는 배열을 이용하고 있으며 본 연구에서는 해쉬 값과 함수의 포인터로 구성되어 있는 배열을 이용하고 있다. 배열을 이용하여 네이티브 함수 테이블을 구현할 경우 메모리 점유율이 높은 단점은 있지만 API에 가상함수를 추가하여 API를 확장할 경우 네이티브 함수 테이블의 재구성이 손쉽다는 점과 속도가 빠르다는 이점이 있다.

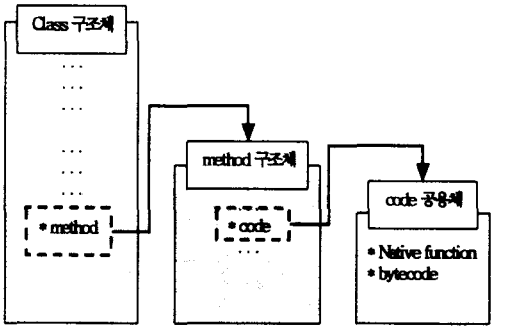
본 연구에서 제안하고 설계한 네이티브 함수 테이블은 [그림 4]와 같은 구조를 가지고 있으며 제한된 자원을 가진 모바일 장치 등에서 적절하게 사용될 수 있다. 특히, 메모리가 부족한 Palm OS 환경에 적합하게 아주 적은 양의 메모리를 점유하도록 설계되어 있다. 가상 함수를 파악하기 위해 정보(package name, class name, method name, signature)들을 모두 저장하지 않고 이 자료들을 가지고 해쉬 값을 구한 후 해쉬 값을 해쉬 테이블의 인덱스로 사용하지 않고 배열에 해쉬 값을 넣어 배열을 검색 하는 방식을 사용하고 있다. 하지만 검색이 이루어지는 동한 매번 해쉬 값을 구하는 작업 및 모든 가상함수의 리스트가 열거되어 있어서 시간적으로는 테이블의 효율이 상당히 떨어지는 단점과 테이블에 함수를 추가하기 위해서는 필요한 정보들을 모두 모아서 계산을 해서 테이블을 재구성해야 한다는 문제점을 가지고 있다.



[그림 4] 네이티브 함수 테이블 구조

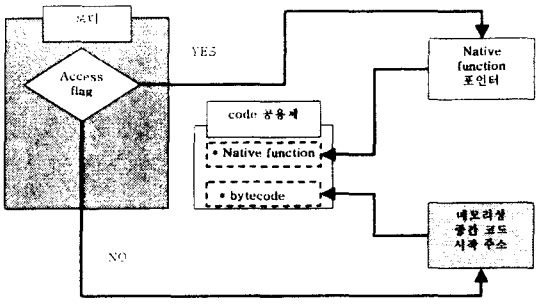
3.2 네이티브 함수 로딩 구조

네이티브 함수의 적재는 로더에서 발생하게 되며 Java언어의 경우 "native"라는 액세스 플래그에 의해서 가상 함수인지 일반 함수인지 구별을 하게 된다. "native" 액세스 플래그는 가상기계의 내부적에서 로딩 과정과 실행 과정에서 가상함수의 구분이 필요 있을 때 사용하는 플래그이다. 네이티브 함수에 대한 적재 방식은 JVM, KVM과 동일한 방식으로 이루어지고 있으며 로더가 클래스 파일을 저장할 수 있는 [그림 5]와 같은 class 구조체에 중간 코드를 적재하는 과정에서 이루어지고 있다.



[그림 5] class 구조체 형태

class 구조체는 중간 코드에 대한 모든 정보를 저장하고 있으며 매소드 단위로 각각의 구조체를 이용하여 메소드에 관한 정보를 저장하고 있다. 메소드 구조체에 들어가는 정보는 매개변수, 반환 값, 플래그 정보 등이 포함되어 있다. 실제 네이티브 함수의 로딩 과정은 [그림 6]과 같이 로더가 엑세스 플래그를 검사한 후 code 정보에 관한 공용체를 이용하여 가상함수일 경우는 네이티브 함수 테이블에서 네이티브 함수에 관한 포인터를 얻어와서 로딩하며 일반 함수일 경우에는 중간 코드의 시작 주소를 적재하여 인터프리터가 이를 수행 할 수 있도록 하고있다.

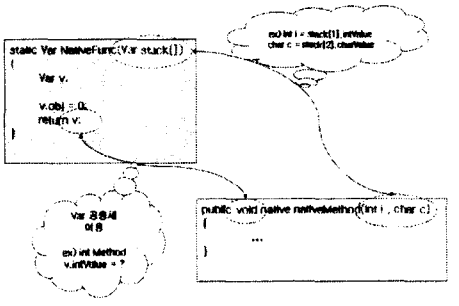


[그림 6] 네이티브 함수의 로딩 과정

3.3 네이티브 함수 실행 구조

네이티브 함수의 실행은 인터프리터에서 중간 코드에 대한 작업을 수행 중에 네이티브 억세스 플래그를 만나게 되면 로딩시 code 공용체에 저장된 네이티브 함수의 포인터를 이용하여 해당 함수를 호출하게 된다.

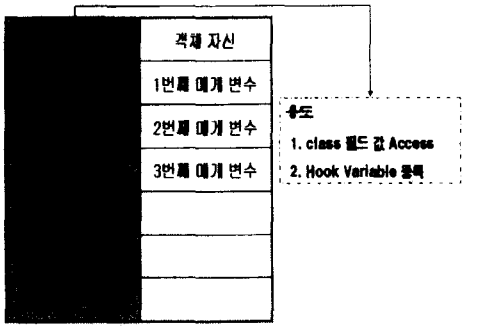
본 연구에서 구현한 네이티브 함수 호출을 위한 인터페이스는 인터프리터에서 호출시 매개 변수 등 필요한 정보들을 실행 스택(execute stack)에 로딩되는 시점의 시작 주소 포인터를 넘겨주며 네이티브 함수는 매개 변수로 넘겨받은 stack[]를 이용하여 인터프리터와 정보를 주고받을 수 있게되어 있다. 실제 가상기계에서 구현된 네이티브 함수의 호출 순서 및 구조는 [그림 7]과 같다.



[그림 7] 네이티브 함수의 호출 순서 및 구조

네이티브 함수 인터페이스는 모든 자료형을 저장할 수 있는 var 공용체를 통해 가상함수의 매개변수 및 반환 값을 처리한다. 가상함수의 매개변수에 관한 데이터는 인터프리터의 실행 스택에서 해당 시작주소를 stack[]에 저장해 전달함으로써 배열에서 사용하는 것처럼 사용할 수 있게 되며 모든 작업을 마치고 나면 기존의 가상함수의 반환 값에 적합한 자료 형을 var형 변수에 저장하여 반환하는 구조로 구성되어 있다.

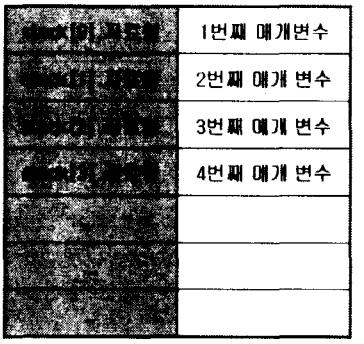
비정적 가상함수(non-static virtual function)의 구조는 객체에 대한 정보를 필요로 하고 있다. 비정적일 경우 객체에 대한 정보를 필요로 하는 이유는 가상 함수가 객체의 멤버로써 작동을 해야 하기 때문이며 실행 스택에는 객체에 대한 정보가 저장되기 때문에 사용을 하지 않더라도 반듯이 객체의 값까지 팝(pop)해야 한다. 비정적 가상 함수 연결을 위한 var stack[] 구조는 [그림 8]과 같다.



[그림 8] 비정적 가상함수 연결을 위한 var stack[] 구조

정적 가상함수(static virtual function)의 경우는 클래스 내부에 포함 되어 있지만 별로의 메모리 공간을 가지고 있는 함수로써 객체에 종속적이 않고 함수 자체적으로 기능을 할 수 있는 함수이다. 이러한 정적 가상함수는 네이티브 함수 구현에 있어서 객체에 대한 정보가 필요하지 않으며 매개 변수만 받아 연산을 수

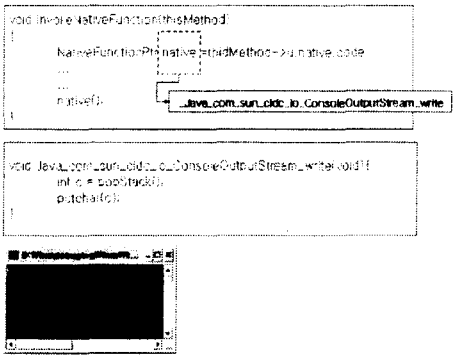
행한 후 결과를 반환한다. 정적 가상 함수 연결을 위한 var stack[] 구조는 [그림 9]와 같다.



[그림 9] 정적 가상함수 연결을 위한 var stack[] 구조

3.4 정적/동적 네이티브 함수 실행

가상함수가 정적/동적 네이티브 함수 연결을 통해서 실질적으로 실행되는 과정은 [그림 10]과 같다. HelloWorld.java 프로그램이 실행되는 중간 과정이며 이 프로그램에서 사용하는 println()메소드는 가상함수인 write() 메소드에 캐릭터 문자 하나씩 전달하며 호출하는 구조로 이루어져있다. 이러한 write() 메소드는 표준 출력을 수행하는 가상함수이기 때문에 C언어 함수인 putchar()로 구현된네이티브 함수를 호출함으로써 화면상에 문자를 출력하게 된다.



[그림 10] 정적/동적 네이티브 함수 실행 결과

4. 결 론

가상기계란 프로세서, 운영체제 등이 바뀌더라도 응용 프로그램을 변경하지 않고 사용할 수 있는 기술로 특히, 임베디드 시스템을 위한 가상기계 기술은 모바일 디바이스와 디지털 TV 등에 탑재할 수 있는 핵심기술로 다운로드 솔루션에서는 꼭 필요한 소프트웨어 기술이다.

본 연구에서는 임베디드 시스템에서 위한 가상기계의 구축 시에 필요한 최적의 실행 환경을 지원하기 위해 정적 및 동적 라이브러리 연결에 기법을 연구한다. 특히, 동적 라이브러리 연결에 관한 연구는 보다 효율적인 응용프로그램의 실행을 위해 최근 사용되는 고급 언어에서 반드시 요청되는 기술로서 이를 지원하기 위한 국내외 연구가 활발히 진행되고 있다.

본 연구를 위해 기존의 JVM, KVM에 대한 상세 분석 연구를 진행하였으며 특히, 정적/동적 라이브러리 연결 기법을 조사하기 위해 실질적인 소스 분석 연구를 진행하였다. 다양한 기초 연구를 기반으로 실제적으로 가상기계에서 라이브러리 연결에 필요한 네이티브 함수 구조, 네이티브 함수 연결 기법, 네이티브 함수 연결을 위한 네이티브 함수 테이블을 구현하였다. 또한 구현된 네이티브 함수 테이블 및 연결 기법을 검증하고 분석하기 위해 실험 연구를 진행하였다.

향후 보다 세부화된 네이티브 함수 연결 기법에 관한 알고리즘 개발 연구가 진행되어야 하며 이를 보완하기 위한 연구가 진행중이다. 또한 보다 다양한 실험 연구를 통해 검증 부분이 보완되어야 한다.

참 고 문 헌

[1] Alfred V. Aho, Mahadevan Ganapathi,

Steven W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," ACM TOPLAS, Vol. 11, No. 4., pp.491-516, Oct., 1989.

[2] Christopher W. Fraser, David R. Hanson, Todd A. Proebsting, "Engineering a Simple, Efficient Code Generator", ACM Letters on Programming Languages and System, pp.231-226, September, 1992.

[3] 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.

[4] Ken Arnold, James Gosling, "The Java™ Programming Language", Addison Wesley, 1997.

[5] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, 2/E, Addison Wesley, 1999.

[6] Jon Meyer, Troy Downing, Java Virtual Machine, O'Reilly & Associates, 1997.

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

[8] Jason Hunt and Ron Cytron, "Java Bytecode Assembler", Washington Univ., 1996.

[9] Eric Gunnerson, A Programmer's Introduction8 to C#, ApressTM, 2000.

[10] ECMA, Standard ECMA-334 C# Language Specification, 2001.

[11] ECMA, Standard ECMA-335 Common Language Infrastructure(CLI), 2001.

[12] Microsoft, .NET, C# & ASP.NET Programming, .NET Press, 2001.

[13] Tom Archer, Inside C#, Microsoft press, 2001.

[14] Microsoft, MSIL Instruction Set Specification", Nov. 20. 2000.

[15] Microsoft, "The IL Assembly Language Programmer's Reference", Oct. 2000.

고 광 만

1991. 02. 원광대학교 컴퓨터공학과 졸업
1993. 02. 동국대학교 컴퓨터공학과 석사학위 취득
1998. 02. 동국대학교 컴퓨터공학과 박사학위 취득
1998. 03. ~ 2001. 08. 광주여자대학교 컴퓨터학부 전임강사
2002. 12. ~ 2003. 02. Queensland University of Technology 연구교수
2001. 09. ~ 현재 상지대학교 컴퓨터정보공학부 조교수
관심분야 : 프로그래밍 언어 설계 및 구현