가상기계 탑재를 위한 재목적 어댑터 설계†

(Retargetable Adapter Design for the Virtual Machine Porting)

고 광 만*
(Kwang-Man Ko)

요 약 본 논문에서 재목적 기술을 이용하여 임베디드 시스템상에 가상기계를 효과적으로 탑재할 수 있는 방법을 설계한다. 이를 위해 가상기계를 플랫폼 독립적인 요소와 플랫폼 의존적인 요소로 구분한 후 플랫폼 의존적인 부분을 재목적 기술을 적용하여 어댑터를 설계하고 구현하였으며 이를 증명할 수 있는 정형화된 방법을 제시하였다.

핵심주제어 : 재목적 기술, 가상기계, 어댑터, 임베디드 시스템

Abstract In this paper, We have defined an effectual way to adapt the virtual machine into an embedded system, and mobile device platform using retargetable technology. For this technology, We classified platform dependent character, and platform independent character, then designed, and implemented the adapter by adopting platform dependent character to retargetable technology, which allows to express in a formal method.

Key Words : Retargetable Technology, Virtual Machine, Adapter, Embedded System

1. 서 론

최근 인터넷 및 무선 통신 기술이 급속도로 발전되고 응용 분야가 확대되면서 응용 소프트웨어를 프로세서나 운영체제와 같은 플랫폼에 의존하지 않고 실행할 수 있는 기술에 대한 연구와 이를 지원할 수 있는 실행시간 환경 개발에 관한 연구가 국내외적으로 활발하게 진행되고 있다. 가상기계[1][2]는 에플리케이션이 실행되는 환경인 플랫폼이 변경되더라도 에플리케이션의 변경없이 실행될 수 있는 환경을 지원한다. 하지만 가상기계를 다양한 플랫폼에 탑재하기 위해서 가상기계 개발자는 다양한 플랫폼에 적합한 가상기계를 별도로 개발해야 하는 부담을 갖는다[3]. 본 논문에서는 재목적 기술(retargetable technology)을 적용하여 가상기계를 다양한 플랫폼에 편리하게 탑재할 수 있는 모델을 제시하고 이를 위해 다양한 플랫폼의 특성을 고려하여 실질적인 가상기계 탑재를 담당하는 어댑터를 모델을 설계하고 이를 구현한다.

† 이 논문은 2006년도 상지대학교 교내 연구비 지원에 의한 것임
* 상지대학교 컴퓨터정보공학부

2. 관련 연구(Related Works)

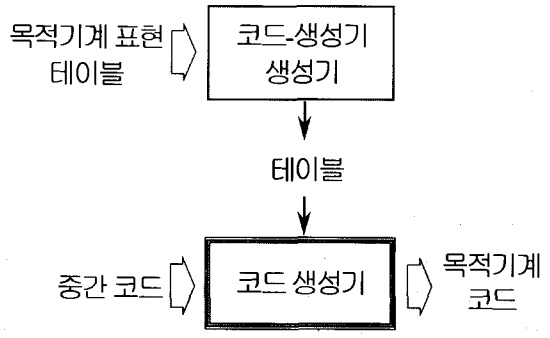
2.1 재목적 기술

재목적 기술은 컴파일러 후단부 개발시에 정형화된 방법으로 중간 코드로부터 다양한 목적기계 코드를 자동 생성하는 기술로서 컴파일러 후단부의

핵심인 목적기계 코드 생성기의 개발을 보다 쉽게 해주며 다양한 목적기계에 적합한 코드를 용이하게 생성할 수 있는 특징을 가지고 있다[11].

재목적 기술은 다양한 목적기계 코드를 정형화된 방법으로 자동 생성할 수 있는 재목적 코드 생성 시스템에서 전통적으로 활용되어 왔다. 또한 재목적 기술을 적용하여 코드 최적화를 위한 분야에도 적용되고 있다. 최근에는 임베디드 프로세서를 위한 저수준의 기계 코드를 생성하는 분야에서 활발하게 적용되고 있다. 실질적으로 임베디드 시스템은 운영체제 및 프로세서의 특성이 기존의 PC, 워크스테이션 환경과는 다르게 종류 및 특성이 매우 다양하다. 따라서 특정 임베디드 프로세서에 적합한 목적기계 코드를 생성하기 위해 컴파일러를 개발하는 것은 엄청난 비용이 요구된다. 재목적 기술을 적용하여 다양한 플랫폼 환경에 적합한 목적기계 코드를 생성하는 것은 매우 효율적인 기법으로 간주되고 있다[12].

재목적 코드 생성 시스템은 [그림 1]과 같이 목적기계 표현 테이블, 코드-생성기 생성기, 코드 생성기로 구성되어 있다[13].



<그림 1> 재목적 코드 생성 시스템

목적기계 표현 테이블에는 특정 목적기계에 대한 정보 및 중간 코드에 대한 목적 코드 생성 규칙을 기술한다. 코드-생성기 생성기는 목적기계 표현 테이블을 입력으로 받아 코드 생성시에 참조될 수 있는 정보를 출력한다. 코드 생성기는 중간 코드를 입력으로 받아 실질적으로 목적기계 코드를 생성하는 부분으로서 하나의 공통된 코드 생성 알고리즘을 이용한다. 재목적 컴파일 기법을 이용한 ACK(Amsterdam Compiler Kit)는 EM(Encoding Machine) 중간 코드로부터 목적기계 코드 생성에 필요한 정보를 목적기계 표현 테이블에 정형화된 방법으로 기술하며 코드-생성기 생성기에 의해 목적기계 코드 생성시에 필요한 정보를 생성한다. 코드 생성기는 생성된 정보를 참조하여 중간 코드에 대한 목적기계 코드를 생성한다[14][15].

목적기계 표현 테이블에 기술되는 정보는 목적기계 정보, 스택 정보, 코드 생성 정보로 구분되며 다음과 같이 11개의 부분으로 구성되어 있다.

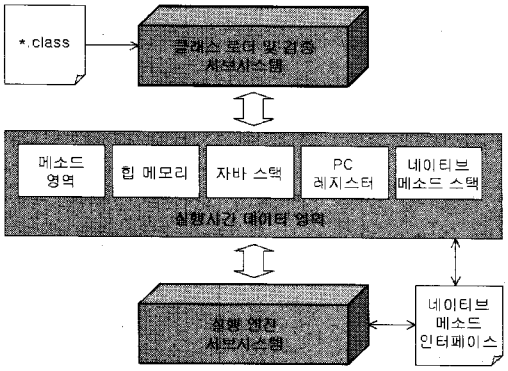
① 상수(constant) 정의 : 워드 및 포인터 크기 등의 필요한 상수 정의
② 속성(property) 정의 : 레지스터를 분류하기 위한 속성 정의
③ 레지스터(register) 정의 : 레지스터 명칭과 속성 정의 부분에서 정의된 속성에 대한 상호 관계 정의
④ 스택 토큰(stack token) 정의 : 코드 생성시 가상 스택에 존재할 수 있는 토큰 정의
⑤ 토큰 집합(sets) 정의 : 주소 표현 방식을 하나의 명칭으로 나타내기 위해 토큰을 원소로 하는 집합 정의
⑥ 명령어(instruction) 정의 : 명령어 명칭과 피연산자로 가능한 토큰 집합, 피연산자의 성질 및 비용 등을 정의
⑦ 토큰 이동(move) 규칙 정의 : 토큰이 현재 명령어의 주소 표현 방식과 일치하지 않을 경우 토큰값 간에 값을 이전시키는 방법 정의
⑧ 테스트(test) 규칙 : 주어진 토큰 집합에 속하는 토큰의 값을 테스트하기 위한 방법 정의
⑨ 스택간 이동(stacking) 규칙 : 가상 스택에 있는 모든 토큰을 실제 스택에 옮길 때 필요한 목적 코드를 만드는 방법을 정의
⑩ 변경(coercions) 규칙 : 가상 스택이 코드 규칙의 스택 패턴과 일치하지 않을 때 스택의 내용을 변경시키는 방법 정의
⑪ 코드 규칙(code rule) : EM 코드 패턴에 대한 목적 코드로의 변환 규칙 정의. EM 코드 패턴에 대한 목적 코드로의 변환 규칙은 지정어 "pat" 다음에 하나 이상의 EM 명령어 패턴과 각 명령어에 대한 피연산자 조건이 기술된다. 스택 패턴은 지정어 "with" 다음에 기술되며 가상 스택에 존재할 수 있는 토큰을 정의한

다. 레지스터 배정은 지정어 "uses" 다음에 특정 레지스터를 기술한다. 실질적으로 출력되는 목적 코드는 지정어 "gen" 다음에 기술된다. 또한 가상 스택에 존재하는 토큰을 변경하기 위한 명령, EM 패턴 대치 명령, 스택간 이동 명령으로 구성되어 있다.

실제로 가상기계가 탑재되는 플랫폼의 특성을 기술하는 플랫폼 디스크립션은 가상기계가 탑재되는 플랫폼 특성을 정형화된 문법을 이용하여 기술하는 부분으로서 재목적 코드 생성 시스템의 목적기계 표현 테이블에 해당된다[16][17][18].

2.2 가상기계에 관한 연구

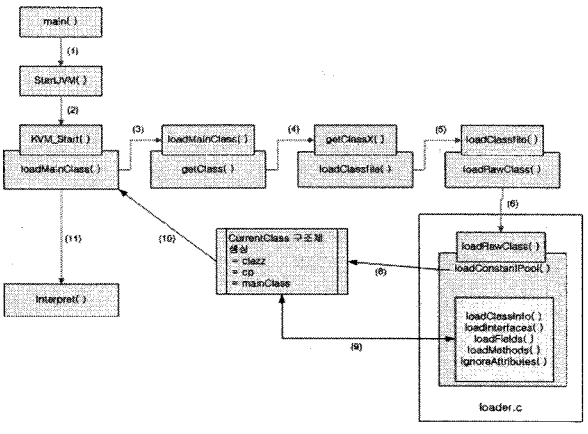
자바 가상기계(Java Virtual Machine)는 자바 프로그램을 실질적으로 실행하는 엔진으로서 231개의 바이트코드 명령어 집합과 클래스 로더 및 검증기, 메모리 관리자, 에러 및 예외 처리기, 네이티브 메소드 링커, 인터프리터 등으로 [그림 2]와 같이 구성되어 있다.



<그림 2> 자바 가상기계의 구조

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 API(Application Programming Interface)의 지원이 요구된다[4][5].

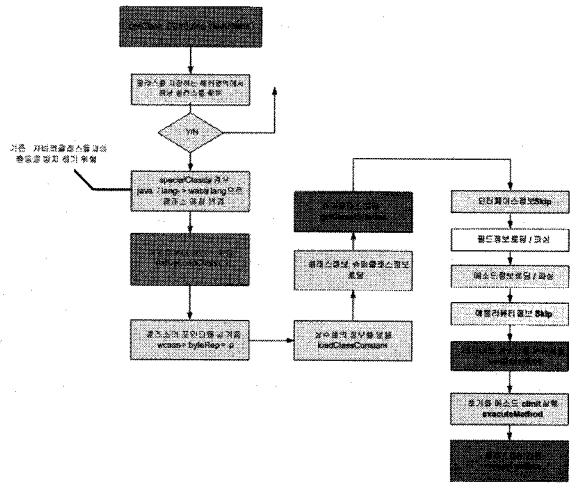
J2ME(Java 2 Micro Edition) 플랫폼은 J2SE(Java 2 Standard Edition), J2EE(Java 2 Enterprise Edition)와 달리 다중 사양을 수용하고 있으며 특히, 메모리 요구 사항에 따라 CLDC(Connected Limited Device Configuration)와 CDC(Connected Device Configuration)로 구분된다. CLDC는 KVM(Kilobyte Virtual Machine)을 가상기계로 채택하고 있다. 이러한 KVM은 작고 한정된 기능을 가진 디바이스에 적합한 콤팩트한 JVM을 구축하기 위해 Palm® 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(Java Native Interface), 리플렉션, RMI(Remote Method Invocation), 객체 직렬화 등을 지원하지 않는 단점을 가지고 있다. 실제 KVM에서 에플리케이션이 실행되는 구조는 [그림 3]과 같다[6][7].



<그림 3> KVM 실행 구조

Waba VM은 Palm®, Windows CE등 소형 장치들을 위해 개발된 자바 가상기계 실행환경으로서 자바의 문법을 따르고 있으며 JVM의 실행 포맷인

클래스 파일의 정보를 이용하여 작동한다. 하지만 기존의 JVM이 사용하던 API를 사용하지 않고 새로 작성된 API를 이용하며 기존 것과는 호환이 되지 않는다는 특징을 가지고 있으며 현재 Windows 계열, Windows CE, Palm® OS용으로 개발되어 있다. Waba VM은 다양한 플랫폼에 탑재되기 때문에 가상기계의 코어부분과 플랫폼에 영향을 받는 포팅 관련 두 부분으로 나누어 구현하게 된다[8]. 따라서 어느 시스템이던지 상관없이 가상기계의 처리 부분은 변하기지 않으며 실제 포팅에 관련되는 부분만 분리하여 작성하므로 기계에 이식성을 증가시킬 수 있다. 실제 Waba VM에서 에플리케이션이 실행되는 구조는 [그림 4]와 같다.

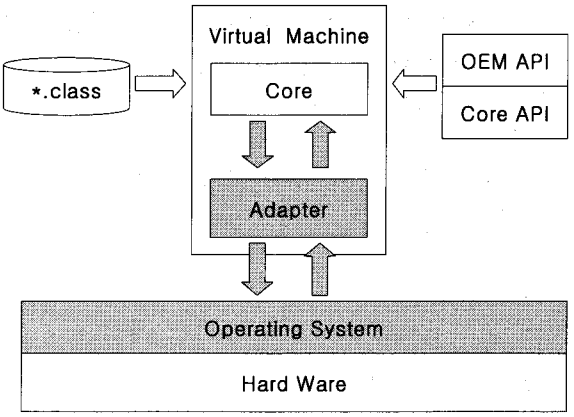


<그림 4> Waba VM 실행 구조

2.3 가상기계 탑재에 관한 연구

가상기계 구현은 가상기계의 핵심 동작을 구현하는 부분과 특정 플랫폼과 인터페이스 부분을 담당하는 어댑터(adapter) 기능을 구현하는 부분으로 [그림 5]와 같이 구분된다[9][10].

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 탑재 할 때마다 새



<그림 5> 가상기계 코어와 어댑터의 관계

롭게 설계하고 구현하는 작업은 상당히 비효율적인 방법이며 특히, 가상기계 개발시에 이식성을 충분이 고려해서 설계 및 구현되어야 한다.

가상기계 코어 부분은 실행 파일 포맷을 로딩하는 로더와 실질적인 실행을 담당하는 인터프리터로 구성되어 있다. 가상기계를 플랫폼에 탑재하기 위한 어댑터는 플랫폼과 가상기계 코어간의 인터페이스 역할을 하며 세부적인 구조는 첫째, vmCore에서 플랫폼 독립적이기 위해 별로 요구하는 자료형을 맞추어 주는 정의 부분, 둘째, vmCore의 입출력과 실행을 위한 인터페이스를 제공하는 주 어댑터 부분, 셋째, 운영체제와 가상기계간에 실질적인 인터페이스를 담당하는 네이티브 코드 부분으로 구성되어 있으며 네이티브 코드 부분의 구현이 가장 큰 비중을 차지한다. 이러한 네이티브 함수는 네이티브 함수 테이블(native function table)을 통해 가상 함수와 매핑될 수 있으며 가상 함수를 추가하고 네이티브 함수 테이블과 네이티브 코드부를 재구성함으로써 가상기계가 지원하는 기능을 계속해서 확장시킬 수 있다. 또한 가상기계의 속도가 최우선 된다면 모든 API들을 네이티브 함수로 구현함으로써 크기는 커지지만 상당히 빠른 실행 속도를 얻을 수 있다.

가상기계의 로더에서 파일을 메모리에 적재하는 도중에 가상함수를 만나게 되면 파일에 있는 코드 대신 네이티브 코드를 저장하고 있는 네이티브 함수의 포인터를 적재하게 되며 가상 함수와 네이티브 함수의 매핑 관계는 네이티브 함수 테이블에 명

시 되어있다. 네이티브 함수 테이블을 통해 가상 함수와 1:1 매핑하는 함수를 네이티브 함수라 부르며 네이티브 함수에서 실질적인 시스템 호출이 일어나게 되므로 가상기계의 네이티브 코드 실행을 위한 부분은 네이티브 함수와 네이티브 함수 테이블로 나누어진다. 네이티브 함수는 가상기계 상의 언어에서 가상 함수를 대신 하는 가상기계 내부에 있는 함수로서 시스템 함수 호출뿐만 아니라 다른 나머지 함수들도 가상 함수를 통해 네이티브 함수를 호출하는 형태로 구현될 수 가있으며 네이티브 함수에 의존도가 높아질수록 가상기계의 속도는 증가하지만 부피는 커지게 된다. 그러므로 반드시 필요한 시스템 호출 함수만 네이티브 함수로 구현하는 것이 보편적이며 이를 통하여 가상기계가 특정 플랫폼에 탑재 될 수 있다. 네이티브 코드로 구현해야 할 부분들은 프로그램이 수행되기 위해서 시스템에 종속적일 수밖에 없는 부분으로서 기본적인 입출력, Network, GUI, 디바이스 요청 등과 같은 부분을 수행해야 한다.

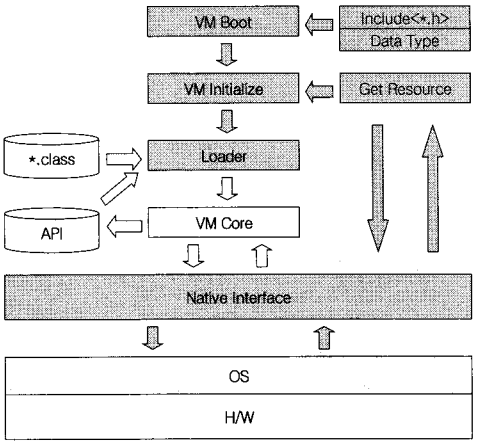
3. 어댑터 설계 및 고려 요소

3.1 어댑터의 설계

가상기계의 탑재를 위해서는 가상기계를 특정 시스템에 적합하도록 개발해야 한다. 자바 프로그램은 특정 플랫폼에 독립적이지만 자바 프로그램을 실행하는 JVM 자체는 플랫폼에 의존적이므로 특정 플랫폼에 적합하도록 개발되어야 한다. 이렇게 특정 시스템에 적합하도록 가상기계를 개발하면 개발 과정에서 공통적인 부분을 발견할 수 있다. 인터프리터와 시스템 호출 함수를 사용하지 않고 작동되어지는 부분은 비슷한 수준의 시스템이라면 모든 플랫폼에 가상기계의 변경없이 이식할 수 있다. 이러한 공통된 부분을 별도로 작성하여 가상기계의 코어를 구현한다.

가상기계의 코어 부분을 제외한 나머지 부분이 코어와 운영체제를 연결하고 코어에서 전달되는 코드를 시스템 고유 함수인 네이티브 코드로 번역해 주는 동작을 어댑터가 수행한다. 따라서 어댑터를 이용하여 가상기계는 장치 독립성을 보장받을 수 있으며 이러한 특성을 정형화된 방법으로 표현하여 재목적성을 가질 수 있다.

어댑터 구조는 가상기계의 크기, 이식성, 속도 차이를 결정할 수 있는 핵심 요소로서 네이티브 인터페이스를 위치를 고려해야 한다. 주 어댑터와 코어를 직접 그 장치의 네이티브 코드로 작성하게 되면 가상기계는 그 크기가 줄고 속도가 향상된다. 또 다른 방법은 주 어댑터와 코어에서 사용되어지는 시스템 함수들을 나중에도 사용할 수 있도록 가상으로 만드는 것이고 네이티브 코드로 작성되어진 함수들을 이용하되 최대한 적게 만들어서 가상 함수들을 최대한 이용해서 작성하는 것이다. 그렇게 되면 다른 장치로 이식할 때도 몇 개의 네이티브 함수 부분만 작성한다. 이와 같은 방법으로 가상기계를 개발하면 크기가 늘고 함수의 호출이 늘어나지만 이식성이 증가되어 다양한 장치로의 탑재가 훨씬 용이하게 된다. 따라서 [그림 6]과 같이 어댑터는 코어를 포함하는 형태가 된다.



<그림 6> 어댑터 구성 요소의 코어간의 관계

3.2 어댑터 구성요소의 주요기능

가상기계를 플랫폼에 탑재하는 주체는 어댑터이며 어댑터는 가상기계를 특정 플랫폼에 이식했을 때 운영체제상에서 정상적으로 수행 될 수 있는 응용 소프트웨어로 만들어 주는 기능을 수행한다. 어댑터는 기초적인 설정들을 위한 정의 부분(Define), 운영체제와 가상기계의 연동을 위한 주 어댑터(Main Adapter), API를 처리하기 위한 네이티브

인터페이스 부분으로 [표 1]과 같이 구성되어 있다.

<표 1> 어댑터의 핵심 구성요소

Adapter	Define	Main Adapter	Native Interface
세부요소	header file, Data type, Endian, System Resource, Flag	Main(), Loader, UI, VM Setup (Boot, Initialize)	Native Code Translater

정의 부분은 자료형 등의 기초적인 설정 사항을 정의하는 부분으로 모바일 장치 등의 특성을 정의하게 된다. 이 부분은 가상기계 제작과 탑재를 위한 기초가 되는 부분이다.

① 가상기계를 탑재할 특정 플랫폼의 SDK에 정의되어 있는 헤더 파일을 지정한다.
② 기본 자료형 지정 - 부동 소수점을 사용할 수 있는지 등등의 여부와 자료형의 크기 제한 등을 정의한다.
③ BIG_ENDIAN과 LITTLE_ENDIAN과 같은 데이터의 저장 형태를 지정한다.
④ 메모리 크기 등의 시스템 자원들의 상황을 얻어내거나 지정한다.
⑤ 장치에서 사용되어질 여러 종류의 플래그들을 지정하고 값을 세팅한다.

주 어댑터는 정의 부분에서 지정된 값들을 받아서 코어를 실행하는 가상기계와 어댑터의 핵심 역할을 수행한다. 주 어댑터는 메인 UI, main() 함수, 로더, 초기화의 구성 요소로 세분화 되어있다.

① 메인 UI는 모바일 장치에서 가상기계 제어 윈도우 등의 메인 인터페이스를 만들어서 사용자가 가상기계를 제어할 수 있도록 해준다.
② main() 함수는 장치마다 그 명칭이 다르거나 가상기계 시작 등의 내용이 조금씩 다르기 때문에 장치마다 따로 작성되어지게 된다.
③ 로더는 장치마다 다른 파일 관리 기법이 존재하기 때문에 특정 플랫폼의 파일 시스템에 맞춰서 제작해야 한다.
④ 초기화 또한 기종별로 다르기 때문에 어댑터에 속하게 된다.

네이티브 인터페이스는 가상기계에서 전달받는 함수들과 API에서 사용되는 함수들을 네이티브 코드로 번역해서 운영체제에 전달한다. 가상기계의 핵심 부분에서 사용되어지는 함수들은 동일하게 작성되어야 이식성이 높아지므로 각각의 가상기계 함수들을 특정 플랫폼에 적합하게 처리해주는 기능을 가지고 있다.

3.3 어댑터 개발시 문제점

특정 플랫폼에 가상기계를 탑재하기 위해서는 플랫폼에 대해 기본 지식과 자료가 있어야 하고 플랫폼에 대한 SDK로 프로그램을 작성할 수 있어야 한다. 로더를 제외했을 때 간단한 연산기능을 수행하는 가상기계는 구현이 쉽지만 입출력과 GUI, 하드웨어 제어 등을 할 수 있는 가상기계를 제작하기 위해서는 그 장치의 API를 이용해야 한다.

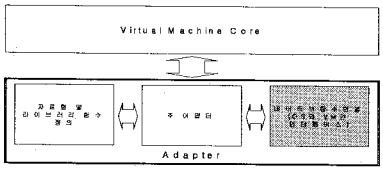
첫 번째 고려사항은 대상 플랫폼에 대한 지식이다. 기계적으로는 장치의 자료 형과 사용가능한 자원 즉, 기억장치나 입출력 장치 등을 고려해야 한다. 소프트웨어적으로는 사용되는 함수들과 프로그래밍 스타일, 파일 관리기법, 데이터의 전송방식 등이 고려되어야 한다. 두 번째 고려사항은 제작한 가상기계가 어느 정도까지 기능을 수행해야 하는가이다. 세 번째 고려사항은 어댑터의 구조이다. 네이티브 코드를 많이 써서 속도를 우선으로 할 것인가 아니면 공통 함수를 만들고 네이티브 코드를 줄여서 크기가 늘고 속도가 약간 떨어지더라도 이식성을 높일것 인가를 결정해야한다.

4. 어댑터 및 API 구현

4.1 어댑터의 세부 구현 모델

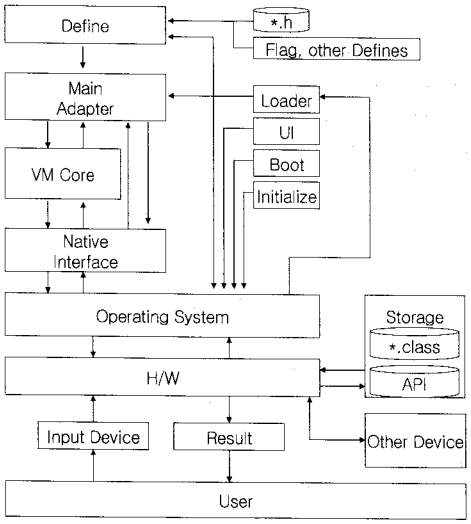
어댑터 구조는 기존의 Waba VM에 기반을 두고 있으며 가상기계 코어의 플랫폼 독립성을 보존하기 위해 플랫폼에 의존적인 부분을 분리한 후 어댑터 구조를 크게 3부분으로 나누어 구현하였다. 어댑터의 구성은 자료형의 크기를 맞추어 주는 정의 부분, 가상기계 코어의 입출력과 실행을 위한 인터페이스를 제공하는 주 어댑터 부분, 마지막으로 운영체제

와 가상기계간에 실질적인 인터페이스를 담당하는 네이티브 함수 부분으로 [그림 7]과 같이 구성되어 있다.



<그림 7> 어댑터 구현 모델

다양한 플랫폼에 가상기계 보다 용이하게 탑재하기 위해 가상기계를 플랫폼 독립적인 부분과 플랫폼 의존적인 부분으로 구분한 후 [그림 8]과 같은 세부 구현 모델을 설계한 후 구현시 반드시 고려해야할 요소와 재목적 요소를 고려하여 실질적으로 어댑터를 구현하였다.



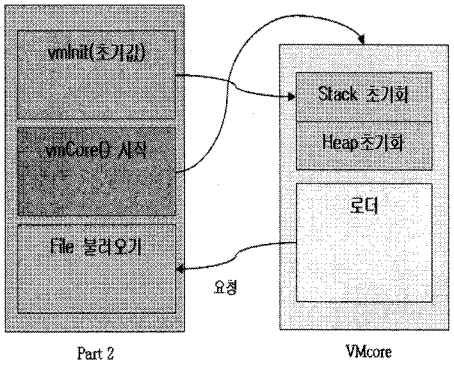
<그림 8> 어댑터의 세부 설계

실제 구현시에 정의 부분에서는 운영체제로부터 입출력 장치와 저장 장치에 대한 정보와 사용 가능한 시스템 자원을 얻는다. 그리고 주 어댑터는 그 정보를 이용해서 가상기계의 실행 준비와 UI를 만들고 클래스 파일의 위치를 알아내고 로더를 불러서 클래스 파일을 읽어온다. 마지막으로 인터프리터를 실행 시킨다. 인터프리터가 바이트 코드를 실행 하다가 출력을 위한 API를 호출하게 되면 로더는 API를 가져오고 네이티브 인터페이스는 해당되는 네이티브 함수를 사용하여 LCD나 CRT 등의 출력 장치를 통해서 문자를 출력한다.

4.2 정의 부분, 주 어댑터, 인터페이스

정의 부분에서는 가상기계에서 사용해야하는 필요한 모든 정보들을 가지고 있다. 컴파일 시 참조할 파일들을 지정하며 장치의 자원 정보, 장치 ID, 기타 사용할 플래그 등을 지정한다. 또한 지원할 자료형, 자료형의 길이 제한, 데이터 저장 방식인 Endian, 저장 공간 용량, 기억장치 용량의 인식, 시스템 시간, 기본 입력 장치에 대한 정보를 주 어댑터로 전달한다.

주 어댑터는 어댑터의 핵심으로서 [그림 9]와 같이 가상기계 코어와 호출 및 반환 관계를 유지한다.



<그림 9> 주 어댑터와 가상기계 코어와의 관계

대부분의 모바일 SDK는 UI를 위한 라이브러리들을 제공한다. 주 어댑터에서는 이러한 라이브러리를 이용해서 UI를 작성하고 모바일 장치의 이벤트 키 값을 지정한다. main() 함수에서는 가상기계의 초기화, 로더의 실행과 코어와의 연동을 통한 인터프리터의 실행을 위한 코드를 가지고 있다. 로더는 장치의 파일 관리기법에 따라서 클래스 파일을 받고 인터프리터의 요청에 의해서 API들을 읽어 들인다. 초기화는 장치의 자원 상황과 플래그의 기본 값들을 정의 부분으로부터 받아 가상기계가 실행 될 수 있는 상태를 만들어 준다. 또한 네이티브 인터페이스를 호출하기 위해서 네이티브 함수들의 인덱스 값을 지정한다.

네이티브 인터페이스는 가상기계의 코어와 주 어

댑터에서 사용되어지는 가상 함수들을 실제로 구현하는 부분으로서 단순한 변환기의 구조를 가지고 있다. 가상 함수, 네이티브 코드, 운영체제의 구조로 되어있다. 가상 함수는 공통적인 것이라서 특정 시스템에서는 추가적인 정보를 요청할 수도 있다. 이 경우에는 네이티브 함수를 구현할 때 추가적인 구현이 필요하다. 주 어댑터가 효과적으로 설계되어 있다면 인덱스 값과 가상 함수의 추가와 네이티브 함수의 추가로 가상기계의 기능을 지속적으로 확장할 수 있다.

4.3 가상기계와 API

가상기계에서 에플리케이션을 개발하는 언어가 플랫폼에 독립적인 특성을 지원하여도 플랫폼에 대한 규격이 변경되면 언어에서 지원하는 API의 구조도 변경해야 한다. 따라서 API들을 다시 작성해 주어야 하는 불편함이 발생한다. 이러한 원인은 장치의 특성과 밀접한 연관을 가지고 있으며 가상기계가 기능 및 성능을 결정하는 요소로 간주된다.그 외에도 개발자의 구현 의도에 따라 다르게 개발될 수 있다. JVM과 KVM, Waba VM은 자바 언어를 지원하는 가상기계이지만 각 가상기계의 API는 서로 다르다. JVM은 PC급 이상의 컴퓨터를 위해서 만들어져 있기 때문에 다른 두 가상기계와 비교했을 때 크기가 크고 기능이 많다. KVM과 Waba VM은 저장 장치가 작고 저 전력, 적은 기능을 위해 개발되었기 때문에 JVM의 API를 그대로 사용할 수 없다. KVM의 경우 JVM의 API를 약간 수정하거나 그대로 사용하지만 모바일 특성 때문에 몇몇 API들을 새로 작성해주어 한다.

실제로 윈도우 플랫폼에서 메모리 사용량을 검사하는 API 함수는 다음과 같은 작성과정을 거쳐 작성된다

① MemoryStatus.java 파일에 native로 선언된 Method 작성

```
//Memorystatus.java
class MemoryStatus {
  public static void main(String args[]) {
  MemoryStatus stat = new MemoryStatus();

  System.out.println("Total Physical Memory=
          "+stat.getTotalPhysical());
  System.out.println("Available Physical
      Memory= "+stat.getAvailPhysical());
  // . . .                    .
  }
  public native long getTotalPhysical();
  public native long getAvailPhysical();
  // . . .
}
```

② 자바 소스를 컴파일

③ JNI 스타일의 헤더 파일을 생성

④ JNI 스타일 헤더 파일을 포함하는 실제 구현 라이브러리 작성(WinMemory.c)

```
//WinMemory.c
#include <windows.h>
#include "MemoryStatus.hi°
JNIEXPORT jlong JNICALL
    Java_MemoryStatus_getTotalPhysical
      (JNIEnv *env, jobject obj)
{
     MEMORYSTATUS memStat;
     GlobalMemoryStatus(&memStat);
     return (jlong) (int)
            memStat.dwTotalPhys;
}
JNIEXPORT jlong JNICALL
    Java_MemoryStatus_getAvailPhysical
    (JNIEnv *env, jobject obj)
{
     MEMORYSTATUS memStat;
     GlobalMemoryStatus(&memStat);
     return (jlong) (int)
            memStat.dwAvailPhys;
}
// ...
```

⑤ 윈도우의 네이티브 라이브러리를 컴파일

⑥ 작성된 winmem.dll 파일을 읽어들이는 것은 MemoryStatus.java 소스 코드에 있는 System.loadLibrary("winmem"); 루틴임

API들은 탑재 영역에서 핵심 요소는 아니다. 하지만 가상기계 환경에서 특정 언어에 대한 프로그램을 실행하기 위해서는 반드시 필요한 부분이다. 특히 시스템 API들은 네이티브 인터페이스를 사용하는 주체가 된다. API들은 자바로 작성되어있고 시스템 호출을 사용하는 부분은 가상 함수 또는 직접적인 네이티브 함수를 호출하게 된다. 물론 이 부분은 가상기계가 API를 실행 할 때 작동되는 부분이므로 API 자체가 호출하는 것은 아니고 인터프리터에서 호출되는 것이다.

또한 API들은 장치의 특성에 맞게 작성되어야 한다. API들 중에서 시스템 API들은 네이티브 인터페이스를 사용할 수 있도록 개발되었다. 시스템 API들을 세부적으로 만들어 놓으면 추가적인 API들은 네이티브 인터페이스에 상관없이 시스템 API들의 사용법만 안다면 쉽게 개발될 수 있다.

5. 결론 및 향후 연구 방향

JVM, KVM, Waba VM과 같은 가상기계에서 수행되는 에플리케이션은 플랫폼의 특성에 의존하지 않는 독립적인 특성을 갖지만 가상기계의 내부적인 다양한 요소에 대해서는 의존적인 성질을 갖는다. 모바일 디바이스, 임베디드 시스템과 같은 다양한 플랫폼에 가상기계를 탑재하기 위해서는 가상기계라는 어플리케이션이 특정 플랫폼의 자원을 충분히 활용할 수 있도록 메모리 크기, 자료형, 입/출력 장치의 인터페이스 등을 고려하여 가상기계를 재구성해야 한다. 특히, 어댑터는 가상기계가 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 플랫폼의 특성인 자료형, 사용가능한 자원, 소프트웨어적으로 지원하는 함수, 파일 관리 기법 등에 대한 기반 연구가 선행되어야 하며 그 내부 동작 원리에 대한 충분한 지식을 갖추고 있어야 한다. 본 논문에서는 기존의 관련 연구를 통해 가상기계 동작 원리 및 어댑터 개발에 필요한 다양한 연구를 진행하였다. 특히, 본 논문에서는 다양한 플랫폼에 가상기계를 재목적 기술을 적용하여 효율적으로 탑재하기 위한 모델을 설정한 후 각 세부 요소를 설계하고 구현하였다. 특히, 임베디드 시스템과 같은 다양한 실행 환경을 갖는 구조에서는 본 논문에서 제시한 어댑터의 기술을 적용하여 보다 용이하게 가상기계를 탑재할 수 있을 것으로 기대하고 있다.

본 연구는 아직 완벽한 실행 환경을 갖춘 임베디드 시스템에 적용하기에는 어댑터 기술과 더불어 다양한 추가적인 요소의 연구 개발이 지속적으로 진행되어야 한다. 이를 보완하여 실질적으로 다양한 산업 분야에서 활용될 수 있도록 할 예정이다.

참 고 문 헌

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

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

[3] Iain D. Craig, Virtual Machines, Springer, 2004.

[4] James E. Smith and Ravi Nair, Virtual machines : Versatile latforms for Systems and Processes, Morgan Kaufmann, 2005.

[5] Etienne M. Gagnon and Laurie J. Henfren, "Sable VM: A Reserach Framework for the Efficient Execution of Java Bytecode", Java Virtual Machine Research and Technology Symposium, 2001.

[6] Efrms G. Mallach, "On the Relationship Between Virtual Machines and emulators", Proc. Workshop on Virtual Computer Systems, Cambridge, MA, pp.117~112, 1973.

[7] Phillip Stanley-Merbell and Liviu Iftode, "Scylla: A Smart Virtual machine for Mobile Embedded Systems", In proc. of the 3rd IEEE Workshop Mobile Computing Systems and Applications, pp.335~339, 2002.

[8] James E. Smith, "An Overview of Virtual Machine Architectures", White Paper, University of Winsconsin-Madison, pp.21~35, 2003.

[9] T. Garfinkel, B. Pfaff, J. Chow, M.

Rosenblum, D. Boneh, "Terra: A virtual machine-based platform for trusted computing", In Proceedings of SOSP 2003, pp.193~206, 2003.
[10] Nik Shaylor, Douglas N. Simon, William R. Bush, "A Java Virtual Machine Architecture for Very Small Devices", LCTES 2003.
[11] D. Bradlee, R. Henry, S. Eagger, "The Marion system for retargetable instruction scheduling", ACM SIGPLAN Conference on Programming Language Design and Implementation, 1991.
[12] R. Leupers, P. Marwedel, "Retargetable Compiler Technology for Embedded Systems : Tools and Applications", Kluwer Academic Publishers, 2001.
[13] Mahadevan Ganapathi, Charles N. Fisher & John L. Hennessy, "Retargetable Compiler Code Generation," ACM Computing Surveys, Vol.14, No.4, 1982.
[14] Andrew S. Tanenbaum, Hans van Staveren and Johan W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," CACM, Vol. 26, No. 9, Sep., 1983.
[15] R. G. G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions," ACM TOPLAS, Vol. 2, No. 2, pp.173-190, April, 1980.
[16] Leupers Rainer, "Compiler Design Issues for Embedded Processors", IEEE Design & Test of Computers, July 2002.
[17] Wahlen O. and Hohenauer, M. and Leupers, R and Meyr, H, "Instruction Scheduler Generation for Retargetable Compilation", IEEE Design & Test of Computers, Feb. 2003.
[18] Rainer Leuper, "An Executable Intermediate Representation for Retargetable Compilation and High-level Code Optimization", In Int. Workshop on Systems, Architecture, Modeling and Simulation, Greece, July 2003.

고 광 만 (Kwang-Man Ko)

- 1991년 2월: 원광대학교 컴퓨터공학과 (공학사)
- 1993년 2월: 동국대학교 컴퓨터공학과(공학석사)
- 1998년 2월: 동국대학교 컴퓨터공학과(공학박사)
- 2001년 8월: 광주여자대학교 컴퓨터과학과(전임)
- 2002년~ 2003년: QUT(호주) 연구교수
- 2001년 9월~현재 상지대학교 컴퓨터정보공학부 부교수
- 관심분야 : 프로그래밍언어론 및 컴파일러