소규모 장치를 위한 가상기계의 설계에 관한 연구

*김선귀, 고광만
상지대학교 컴퓨터정보공학부
e-mail:(skwi78, kkman)@mail.sangji.ac.kr

A Study on the Virtual Machine Design for Small-sized Device

*Sun-Kwi Kim, Kwang-Man Ko
School of Computer and Information Engineering,
Sangji University

요 약

현재 다양한 플랫폼에서 수행되는 가상기계가 개발되고 있으며 특히 소규모 장치들에 내장되어 가고 있다. 소규모 장치는 제한된 시스템 자원을 가지고 있기 때문에 적은 자원을 효율적으로 관리하기 위한 방법이 제시되어야 한다. 본 논문에서는 Palm장치를 위한 가상기계인 Waba를 기반으로 하여 소규모 장치에서 작동되는 가상기계를 설계하였다. 이를 위해 전체 시스템 구조를 설계한 후 실제 실행 과정의 각 단계에서 수행되는 세부 동작을 정의하였으며 실행을 위한 초기화 과정과 인터프리터의 세부 구조에 대해 설계하였다.

1. 서론

최근 하드웨어의 발전으로 컴퓨터 시스템이 아닌 소규모 장치에서도 고급 언어로 작성된 프로그램이 수행되도록 하는 환경이 개발되고 있다. 프로그램이 특정 환경에서 수행되기 위해서는 시스템의 하드웨어와 운영체제에 영향을 받으므로 소스 파일은 특정 기계에 맞는 네이티브 코드를 생성하기 위해 컴파일이 되어야 한다. 이 때문에 시스템 하드웨어나 운영체제의 변경은 새로운 컴파일러의 개발과 API의 변경 및 개발환경의 재구축을 초래하게 된다. 이 문제를 해결하기 위해 현재 가상 기계를 운영체제와 응용 프로그램 사이에 두고 시스템과의 인터페이스를 담당하도록 하고 있다. 또한 컴파일러는 특정기계를 위한 목적코드를 생성하지 않고 가상기계를 위한 중간코드를 만들어 내고 이를 입력으로 받아 수행하도록 한다. 중간 코드로 변경된 프로그램은 해당 중간 코드를 입력으로 하는 가상기계가 내장된 환경에서 다른 변환없이 수행될 수 있게 된다.

현재 다양한 시스템에서 수행되는 가상기계가 개발되고 있으며 점점 소형장치들에 내장되어 가고 있다. 소형장치는 제한된 시스템자원을 가지고 있기 때문에 적은 자원을 효율적으로 관리하기 위한 방법이 제시되어야 한다. 본 논문에서는 Palm장치를 위한 가상기계인 Waba를 기반으로 하여 소규모 장치에서 작동되는 가상기계의 설계하였다. 이를 위해 전체 시스템 구조를 설계한 후 실제 실행 과정의 각 단계에서 수행되는 세부 동작을 정의하였으며 실행을 위한 초기화 과정과 인터프리터의 세부 구조에 대해 설계하였다.

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

2. Waba 가상기계 고찰

2.1 개요

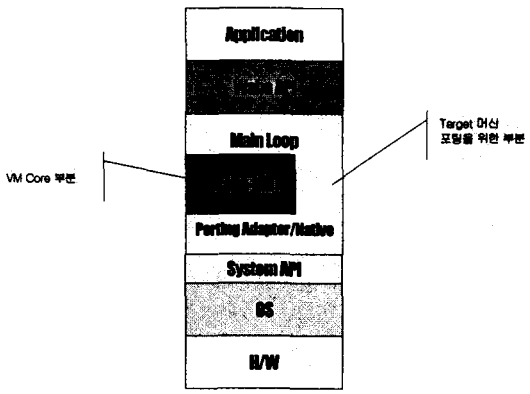
Waba 가상 기계는 Palm, Window CE등 소형 장치들을 위해 개발된 자바 가상기계 실행환경이다[3]. 이것은 자바의 문법을 따르고 있으며 자바 가상기계의 실행 포맷인 클래스 파일의 정보를 이용하여 작동하게 된다. 또한 기존 자바 문법의 서브셋으로 개발되었기 때문에 자바의 문법을 그대로 따르고 있다. 하지만 기존 자바 가상 기계를 위한 API를 사용하지 않고 새로 작성된 API를 이용하며 기존 API와 호환되지 않는다. 현재 Windows 계열, Windows CE, Palm OS용으로 개발되어있으며 각 플랫폼마다

1개의 Waba 핵심(core) 소스와 3개의 포팅(Porting) 관련 소스로 구성되어있다.

Waba에서 구현된 모든 바이트코드는 "Java Virtual Machine"의 명세서를 따르고 있다. Waba에서 정의하고 있는 코드는 총 (0~201) 202개의 바이트코드에 대해 정의하고 있지만 long형, double형, thread, 예외처리 관련 명령어에 대해서는 구현하지 않고 있다(미구현 코드 63개).

2.2 가상기계 탑재 구조

Waba는 Windows 2000/NT/CE와 palm환경과 같이 여러 기종에 탑재되기 때문에 가상기계의 동작부분(Core)과 플랫폼에 영향을 받는 포팅관련 두 부분으로 나누어 구현하게 된다. 따라서 어느 시스템이던지 상관없이 가상기계의 처리 부분은 변하기지 않으며 실제 포팅에 관련되는 부분만 분리하여 작성하므로 기계에 이식성을 증가시킬 수 있다.

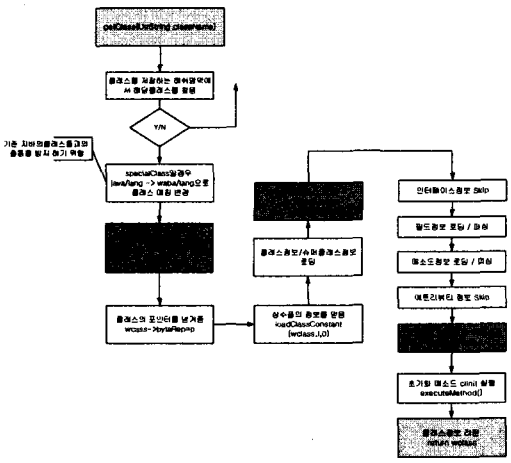


[그림 1] Waba 탑재구조

그림 1과 같이 가상기계의 핵심부분이 존재하며 이를 시스템에 탑재하기 위한 포팅 관련 부분으로 분리되어 있다. 포팅 관련된 부분에는 전체 프로그램의 시작과 종료인 main loop와 해당 운영체제에서 시스템 자원관리 및 처리를 위한 PortAdapter, API Native 함수로 구성되어 있으며 이 부분이 각 시스템마다 다르게 구현되는 부분이 된다. 가상기계의 핵심 부분의 구현은 특정기계를 위한 부분이 아니기 때문에 클래스파일로부터 얻어진 바이트코드의 공통된 기능 처리를 작성하고 해당 운영체제와의 인터페이스처리를 포팅 관련 부분에서 처리하도록 분리하고 있다. 또한 Waba 가상기계에서는 객체의 생성과 메모리관리 산술연산과 같은 기본적인 내부 바이트코드의 처리를 담당하도록 하고 그 외 시스템자원의 처리는 API를 이용하여 수행하도록 하였다. 이때 API는 자바 언어로 작성되어져 있으며 시스템 자원의 관리는 가상기계의 구현시 포팅 관련 함수들과 함께 네이티브 함수들을 구현하고 이를 이 함수들을 API에서 호출 가능하도록 바인딩 시켰다.

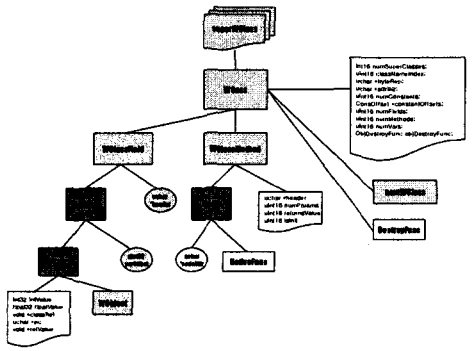
2.3 Waba 가상기계의 세부 분석

가상기계의 실행이 시작되면 실제 스택/힙 메모리 영역을 할당하고 초기화한다. 초기화 과정을 마친후 바이트코드 실행을 위해 그림 2와 같은 클래스 파일의 로딩하는 과정을 거쳐 실행정보로 얻게 된다.



[그림 2] 클래스 파일 로딩 및 실행 정보

로딩과정에서 발생되는 클래스의 실행 정보들은 그림 3과 구조에 저장된다.

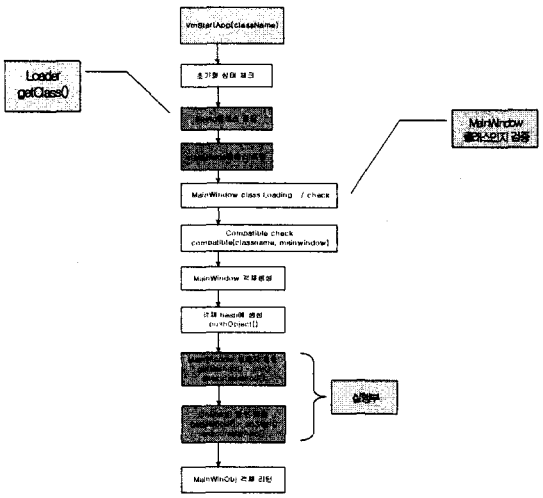


[그림 3] 클래스 파일 정보 저장 구조

실행시 참조되는 정보는 상수풀, 필드, 메소드와 같은 실행 정보를 저장하기 위한 구조로 되어있다. 클래스 로더의 처음 단계는 로딩하고자 하는 클래스가 메모리에 로딩되어 있는지를 체크하고 그렇지 않다면 해당클래스가 시스템 클래스인지를 체크한다. 시스템 클래스일 경우 기존 자바 API 클래스의 로딩을 막기 위해서 Waba API 클래스 경로로 수정하여 준다. 또한 실제 파일로부터 클래스파일을 얻기 위한 nativeClassLoad()를 사용해 클래스 파일을 메모리에 로딩시켜 클래스파일의 시작포인터를 얻은후 클래스파일포맷의 각 오프셋을 이용하여 Constans pool, this class, super class, field, method 정보들을 차례대로 파싱하여 정보를 얻게 된다. 로딩의

최종 단계에서 네이티브 메소드의 호출을 위한 ClassHook()의 바인딩 과정을 거치고 로딩하고자 하는 클래스가 정적 클래스일 경우 cinit()메소드를 실행하여 해당클래스의 정적 필드값을 초기화 시킨다. 이때 각 로딩과정에서 파싱된 정보들은 WClass에 저장된다. 이렇게 로딩된 클래스들은 WClass의 해쉬 구조로 저장되어 있으며 인터프리터 실행시에 참조된다.

그림 4는 클래스 파일에 대해 초기화, 로딩 과정을 거쳐 응용 프로그램이 실제로 수행되는 가상기계의 핵심 부분에 대한 전체 흐름도이다.



[그림 4] 가상기계 실행 과정

3. 소규모 장치를 위한 가상 기계 설계

3.1 전체 개요

가상기계는 중간코드에 저장되어있는 명령어를 실행하기 위한 시스템에 수행되는 하나의 프로그램이다. 본 논문에서는 스택 기반에서 수행되는 가상기계를 구현하기 위해 핵심적으로 수행되는 동작을 5단계로 구분하여 설계하였다.

(1) 가상기계에서 발생하게 되는 에러 메세지를 위한 부분으로 에러 발생 원인을 어떠한 방법으로 처리할지 설정.
(2) 초기화 동작시 필요한 옵션 값들을 가상기계의 초기 실행시 명령어 라인으로부터 받아 처리하는 단계.
(3) 옵션값을 이용하여 가상기계의 실행시 필요한 메모리 영역 할당과 구성 요소들을 초기화하는 단계
(4) 중간 코드를 메모리로 읽어들이고 검증하는 단계, 메모리의 코드 영역에 실행 코드를 저장
(5) 메모리 영역으로부터 실행 코드를 참조하여 각 명령어의 기능을 처리

3.2. 가상 기계 개발의 공통적 요소

매크로 정의 : 구현시 자주 사용하는 명령어들의 집합을 매크로로 정의함으로 문장의 간략화와 코드 이해를 편리하게 할 수 있다.

플랫폼 의존적 코드 : 가상 기계가 수행되는 플랫폼마다 사용되는 메모리 저장 방식, 기본 자료형 등에 대한 차이를 관리하기 위한 함수들의 구현하여 가상 기계에서 올바르게 사용할 수 있도록 변환해 주어야 한다.

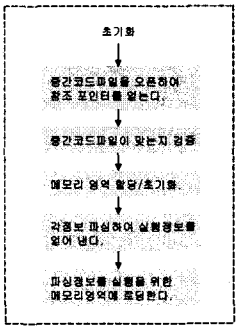
명령어 코드 정의 : 바이트 실행 코드 파일에는 명령어에 대한 표현이 숫자로 되어있다. 가상 기계에서 사용되는 각 코드의 각 명령어마다 니모닉을 붙여준다.

가상 기계 구성 요소 : 가상 기계에서 사용되는 레지스터 변수들과 메모리 참조를 위한 변수들을 정의하며 가상 기계의 실행 엔진에서 참조하여 실행한다.

에러 처리 함수 : 가상 기계의 수행중 발생되는 에러 메시지 처리를 위한 함수를 정의하여 에러 발생시 수행하도록 한다.

3.3 실행 환경 초기화

초기화 과정에서는 메모리 영역 설정과 중간 코드를 위한 실행 정보를 저장하기 위한 자료구조를 설정한다. 객체지향 언어에 대한 가상 기계 개발에서는 클래스 상속 관계를 효과적으로 표현하도록 설계한다. 초기화 과정에서 수행되는 기본적인 사항들은 그림 5와 같다.



[그림 5] 초기화 과정

3.4. 검증

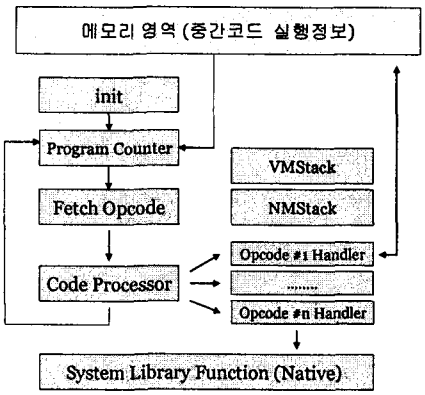
가상 기계의 수행에 있어서 시스템에 오류를 발생시킬 수 있는 코드가 있는지 검증해야한다. 실행 엔진의 처리에 앞서 악의를 가진 명령어 코드나 오류로부터 시스템의 보호하여 안전성을 보장한다. 검증 단계에서 시스템의 안전성을 높이기 위해 다음 사항을 검사한다.

- 중간코드 구조의 명확성
- 정의되지 않은 명령어 코드 검사
- 각 명령어가 참조하는 값(상수/문자)들의 존재 여부

- 각 상수값의 범위 초과여부
- 명령어수행시 스택/지역변수의 상태

3.5 명령어 실행 - 인터프리터

인터프리터의 구현은 가상 기계의 있어서 핵심이 되는 부분으로 중간 코드로부터 얻어진 명령어들을 처리한다. 인터프리터는 명령어를 라인 단위로 처리하기 때문에 하나의 Switch 문장의 구조로 작성되기 쉬우며 전체적인 처리 과정 및 구조는 그림 6과 같다.



[그림 6] 인터프리터 구조

인터프리터는 메모리 영역으로부터 중간 코드를 얻고 이를 처리하여 시스템함수에 의해 결과를 출력한다. 인터프리터에서는 각 명령어마다 수행 해야될 모든 처리가 구현되어 있으며 프로그램 카운터에 의해 해당 명령어 수행 후 다음에 수행되어야 할 명령어을 참조함으로 전체적인 명령어를 처리한다. 수행중 발생되는 결과와 처리 값들은 스택 기반머신의 경우 스택과 힙의 영역에 저장되어 관리되며 각 함수들의 호출 정보들을 스택에 저장한다. 가상기계와 탑제되는 운영체제사이의 처리를 담당하는 시스템라이브러리 함수는 가상 기계의 처리 결과를 해당 기계에 나타내도록 하는 부분이 되며 차후 가상 기계 기반 API 추가시에도 이들 함수의 도움을 받아 시스템 자원에 대한 접근을 처리할 수 있다.

3.7 가상기계 개발시 고려사항

가상 기계는 중간 코드를 입력으로 받아 실행 결과를 생성한다. 따라서 중간 코드가 어떻게 특정기계에서 해석되어야 하는지를 기술하여 주는 것이 가상기계의 개발에 있어서 중요하다. 각 중간 코드의 명령어에 대한 기능은 중간 코드에 대한 명세서에서 정의한 대로 구현하여 주면 되지만 출력 결과를 나타내는 부분은 플랫폼마다 특성이 다르기 때문에 이 부분을 시스템에 맞게 구현해야 한다. 이를 구현하기 위해서 각 운영체제에서 제공되는 시스템 라이브러리함수를 이용하여 그 기능을 구현할 수 있다.

가상 기계에서 중간 코드 정보를 위한 자료구조 : 가상 기계 개발자는 로딩된 중간 코드 정보를 저장하기 위한 메소드 영역, 메모리 구조, 스택 주조 등을 설계해야 한다. 자료 구조를 어떻게 설계하느냐에 따라 실제 실행 엔진과 다른 구성요소들의 구현을 결정할 수 있기 때문이다. 자료구조를 설정하기 위해서는 가상기계의 실행 엔진에서 실행시간에 필요한 정보들이 무엇인지를 결정하고 이를 바탕으로 전체 구조를 정리한다.

로더/링커 : 로더의 구현은 중간 코드로부터 정보를 얻어내고 검증 단계를 거쳐 메소드 영역 등에 실행에 필요한 정보들을 저장하는 역할을 수행하도록 작성한다. 따라서 시스템에 저장된 중간 코드(가상기계 실행 파일 포멧)를 메모리로 이동시키고 검증단계를 거친 후 메소드 영역에 파싱된 정보를 저장하도록 구현한다.

인터프리터 : 인터프리터는 중간 코드의 니모닉 명령어를 라인 단위로 처리하며 수행된다. 따라서 인터프리터의 구조는 니모닉 코드를 위한 반복문 안에 각 코드에 대한 처리를 구현해야 한다. 이 때문에 보통 하나의 커다란 swich 문장 구조로 구현할 수 있으며

4. 결론

가상기계의 목적은 중간코드가 최대한 효과적으로 수행되기 위한 환경을 구축하는 것이다. 이 때문에 중간 실행 파일로부터 실행을 위한 정보를 얻으며 이를 어떻게 구성해야할지를 잘 결정해야하는 것이 무엇보다 중요하다. 가상 기계의 구현하는 개발자는 개발에 앞서 가상 기계의 주요 수행범위와 기능을 계획하고 이를 위한 최적화된 방법을 찾도록 하는 것이 바람직하다. 현재 많은 가상 기계에서는 성능 향상의 방법으로 JIT 컴파일러를 내장하는 방식을 사용하고 있다.

본 논문에서는 기존의 JVM, KVM, Waba 가상 기계를 기반으로 소규모 장치를 위한 가상 기계 개발에 필요한 핵심적인 사항을 분석하고 설계하였다. 특히, 가상 기계 개발시에 공통적으로 개발되어야 하는 부분에 대한 자세한 분석과 이를 통한 모델을 제시하여 향후 개발되는 소규모 장치를 위한 가상 기계 개발에 참조할 수 있도록 하였다.

참고문헌

[1] "Virtual Machine Design and Implementation in c/c++", Blunden
[2] www.wabasoft.com, Waba JVM
[3] "The Java Virtual Machine Specification", Second Edition. Tim Lindholm Frank Yellin.
[4] "자바가상기계 프로그래밍" Engel저 곽용재 역. Addison.