Journal of Korea Multimedia Society Vol. 10, No. 11, November 2007(pp. 1472-1482)

자바 클래스 파일과 .NET PE 파일을 위한 통합 로더/링커 시스템의 개발

고 광 만†

요 약

로더/링커는 자바 클래스 파일 또는 .NET 환경의 중간 표현인 PE 파일을 입력으로 받아 검증, 레졸루션, 초기화, 실행에 필요한 최적화된 정보 저장 등 실질적인 실행에 필요한 모든 정보 생성 및 무결성을 보장하는 아주 중요한 부분이다. 본 논문에서는 자바 클래스 파일과 .NET 환경의 PE 파일에 대한 통합 로더/링커 시스템을 개발하고자 한다. 이를 위해, 자바 클래스 파일과 .NET PE 파일 정보를 모두 저장할 수 있는 새로운 실행 파일 포맷(*.evm) 및 메모리 포맷을 설계했으며 저장된 실행 정보를 활용하여 JVM 또는 .NET 환경에서 실행할 수 있도록 링커/로더 시스템을 구현하였다.

Development of the Integrated Loader/Linker System for the Java Class File and .NET PE File.

Ko Kwang Man†

ABSTRACT

The integrated loader/linker plays a very important role in creating all types of information and ensuring information integrity needed for substantial executions by receiving a PE input file, an intermediate representation of a java class file or a .NET environment, thereby allowing for saving information optimized for verification, resolution, initialization, and execution. This paper proposes a loader/linker system for integrating a java class file and .NET-based PE file. As a means of implementing the loader/linker system, a new execution file format(*.evm) and a memory format were designed to save all information of Java class files and .NET-based PE files, and enable the information in those files to be executed in a JVM or .NET environment through the use of saved execution information.

Key words: Class File(클래스 파일), .NET PE File(.NET PE 파일), Linker, Loader(링커, 로더)

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 활성화되고 있다. 특히, 자바 언어 작성된 프로그램을 플랫폼에 의존하지 않고 실행할 수 있는 환경을 지원하는 자바 가상기계와 마이크로소프트사에서 심혈을 기울이고 있는 .NET 실행시간 환경 기술은 현재 전 세계적으로 응용 분야가 확대되고 있다[1].

하지만 현재까지는 이러한 실행시간 환경에 적합한 컨텐츠 개발에 주력되고 있지만 앞으로는 실행시

※ 교신저자(Corresponding Author) : 고광만, 주소 : 강원도 원주시 우산동 660번지 상지대학교(220-702), 전화 : 033)730-0486, FAX : 033)730-0480,
E-mail : kkman@sangji.ac.kr

접수일 : 2007년 3월 20일, 완료일 : 2007년 10월 19일
† 준회원, 상지대학교 컴퓨터정보공학부 부교수
※ 이 논문은 2005년도 상지대학교 교내 연구비 지원에 의한 것임.

간 환경 개선 및 자바 실행시간 환경과 .NET 실행시간 환경을 통합하여 지원하거나 차세대 기능을 갖춘 새로운 프로그래밍 언어의 출현 및 이를 지원할 수 있는 실행시간 환경 개발에 초점이 모아지고 있다. 특히, 실행시간 환경에 개발에서 핵심이 되는 인터프리터, Just-In-Time 컴파일러와 같은 실행 엔진에 모든 핵심적인 실행 정보를 제공하는 로더/링커의 기술에 대한 중요성 매우 강조되고 있다.

로더/링커는 자바 클래스 파일, .NET 환경의 중간 표현인 PE 파일을 각각 입력으로 받아 이를 실행 환경으로 적재하는 로딩 및 검증, 준비, 레졸루션 동작을 수행하는 링킹 동작을 수행하며 원활한 수행을 위한 초기화, 실행에 필요한 최적화된 정보 저장 등 실행에 필요한 모든 정보 생성 및 무결성을 보장하는 아주 중요한 부분이다[2].

자바 가상기계에서 사용되고 있는 로더/링커 시스템에 관한 표준화된 명세는 지정되지 않고 있으며 실질적으로 자바 가상기계를 개발하는 개발자 및 제공 업체에 따라 기능 및 세부 적인 구조는 약간의 차이점을 가지고 있다. 로딩 과정에서는 입력된 클래스 파일이 자바 가상기계에서 수행될 수 있는 적합한 구조인지를 검사한 후 이를 자바 가상기계 내부에서 사용할 수 있는 내부 자료 구조에 저장한다. 링킹 과정에서는 저장된 자료에 대한 검증 과정을 거친 후 실제 수행을 위한 메모리 초기화 같은 준비 단계를 거친다. 레졸루션 단계에서는 클래스 파일의 상수 풀 정보를 참조하여 심볼릭 참조를 실제 직접 참조로 변환하는 동작을 수행한다. 로더/링커의 마지막 단계인 초기화 단계에서는 실제 수행을 위한 초기화 동작을 수행한다[3,4].

.NET 실행 환경의 로더의 경우에도 메타 데이터와 IL 코드로 구성된 관리 모듈(managed module)을 입력에 대해 실제 수행에 필요한 의미 있는 정보 추출 및 검증을 위해 로더는 관리 모듈의 메타 데이터를 내부 자료 구조로 저장한다. 이러한 정보는 IL 코드를 수행하는 Just-In-Time 컴파일러에 의해 참조되며 이러한 정보를 이용하여 특정 기계에 대한 네이티브 코드를 생성한다.

본 논문에서는 자바 클래스 파일과 .NET 환경의 PE 파일에 대한 통합 로더/링커 시스템을 설계하고 구현한다. 이를 위해, 자바 클래스 파일과 .NET PE 파일 정보를 모두 저장할 수 있는 새로운 실행 파일 포맷(*.evm) 및 메모리 포맷을 설계했으며 저장된 실행 정보를 활용하여 JVM 또는 .NET 환경에서 실행할 수 있도록 링커/로더 시스템을 구현하였다.

2. 기반 연구

2.1 자바 가상기계와 클래스 파일

자바 가상기계는 플랫폼 독립적성을 유지하면 자바 프로그램 실행시켜주는 추상적인 소프트웨어로 작성된 컴퓨터로서 클래스 로더 및 검증 서브시스템, 실행 엔진 서브시스템 및 실행에 필요한 데이터와 이를 저장하고 관리하기 위한 실행 시간 데이터 영역으로 구성되어 있다.

자바 가상기계의 클래스 로더는 시스템 클래스 로더(Bootstrap Class Loader)와 사용자-정의 클래스 로더(User-defined Class Loader)로 구분되며 입력된 클래스 파일에 대한 바이너리 데이터를 자바 가상기계에 로딩, 링킹, 초기화 동작을 순서대로 수행한다. 로딩 단계에서는 클래스 또는 인터페이스와 같은 자료형에 대한 바이너리 데이터를 찾아 적재하는 동작을 수행한다. 링킹 단계에서는 첫째, 적재된 자료형에 대한 정확성을 검증하며 둘째, 클래스 변수에 대한 메모리 할당 및 메모리를 기본값으로 초기화하는 준비 동작을 수행한다. 마지막으로 자료형에 심볼릭 참조를 상수 풀 정보를 이용하여 직접 참조로 변환하는 레졸루션 동작으로 구성되어 있다. 마지막으로 초기화 동작 단계에서는 클래스 변수를 초기화할 수 있는 메소드를 호출하여 적당한 시작 값으로 초기화한다.

메소드 영역에서는 적재된 클래스 또는 인터페이스에 대한 바이너리 데이터로부터 자료형 정보, 상수 풀, 필드 정보, 메소드 정보 등을 추출하여 저장하며 클래스내에 선언된 클래스 변수에 대한 정보도 저장된다. 자료형 정보에는 자료형과 슈퍼 클래스에 대한 패키지 이름, 점 연산자(.), 자료형에 대한 이름으로 구성된 완전한 이름을 저장하며 자료형이 클래스와 인터페이스를 구별할 수 있는 정보와 자료형에 대한 식별자 등도 저장된다.

상수풀 영역에서는 정수, 문자열과 같은 리터널과 클래스 또는 인터페이스와 같은 자료형, 필드, 메소드에 대한 심볼릭 참조에 대한 정보를 배열 구조와 같이 순서화된 집합 구조로 저장한다. 상수풀이 심볼

릭 참조에 대한 정보를 저장하고 있기 때문에 자바 프로그램의 동적 링킹을 수행하는 핵심 역할을 한다.

필드 정보 영역에서는 필드 이름, 필드 자료형 및 필드에 대한 수정자(modifier)에 대한 정보가 선언된 순서에 따라 레코드 형식으로 저장된다. 메소드 정보 영역에서는 메소드 이름, 메소드 복귀형, 매개변수 개수 및 각각의 자료형, 메소드 수정자 등이 선언된 순서에 따라 레코드 형식으로 저장된다. 또한 각 메소드에 대한 바이트코드, 메소드에 대한 스택 프레임에서 오퍼랜드 스택 및 지역 변수 영역의 크기, 예외 테이블이 저장된다. 자료형에 대한 로딩 동작이 사용자-정의 클래스 로더를 통해 수행되면 사용자-정의 클래스 로더의 참조를 메소드 영역의 자료형 정보 영역에 저장한다. 이러한 정보는 애플리케이션 수행 시 동적 링킹 정보로 활용된다. 메소드 영역 구현시에 모든 스레드가 메소드 영역을 공유하므로 모든 스레드가 안전하게 동작할 수 있도록 구현되어야 하며 접근 빠르게 하기 위해 구현 방법에 따라 메소드 테이블을 이용하는 방법에 사용되고 있다.

힙 메모리 영역에서는 자바 애플리케이션에서 새로운 객체가 생성될 때마다 하나의 공통된 힙 영역에서 메모리를 할당한다. 따라서 멀티 스레드가 힙 데이터 영역인 객체에 대한 접근을 효과적으로 진행할 수 있도록 적당한 동기화 방법이 요구된다. 또한 할당된 힙 공간에 대한 사용을 마친 후 반환에 대한 명시적인 명령어를 사용하지 않으므로 이를 효과적으로 관리할 수 있는 가비지 컬렉션이 요구된다.

자바 애플리케이션에서 새로운 스레드가 시작되면 각 스레드의 메소드 호출 및 수행에 따른 정보를 저장하기 위해 스택 프레임을 생성하며 생성된 프레임은 푸쉬와 팝에 의해 관리되는 자바 스택에 저장된다. 각 스레드의 메소드 호출시에 생성되는 스택 프레임은 지역 변수, 오퍼란드 스택, 프레임 데이터 영역으로 구성되어 있다. 지역 변수 영역에서는 메소드 내에 선언된 지역 변수와 매개 변수를 저장한다. 오퍼랜드 스택은 메소드 동작이 수행시에 필요한 값을 관리하는데 사용된다. 프레임 데이터 영역은 동적 링킹을 위한 상수풀 결정, 정상적인 메소드 반환, 예외처리 등에 대한 정보를 저장하고 있다.

클래스 파일은 8비트 단위의 스트림으로 구성되며, 16비트 · 32비트 · 64비트 크기를 가진 데이터들은 8비트 단위로 나누어져 높은 비트가 먼저 나오는 빅 엔디언(Big-Endian)의 순서로 저장된다.

magic 필드는 자바 클래스 파일임을 나타내 주는 식별자로 항상 0xCAFEBABE에 값을 갖는다. minor_version과 major_version 필드는 클래스 파일의 버전을 나타낸다. constant_pool_count는 constant_pool의 개수를 나타내며 항상 0보다 큰 값을 가진다. constant_pool은 가변 크기를 갖는 상수 풀을 표현한 것으로 배열의 형태이며 클래스, 슈퍼 클래스, 메소드, 필드의 이름 등에 대한 정보를 가진다. access_flags는 현재 클래스 파일이 나타내는 클래스의 접근 권한을 나타낸다. this_class는 클래스 파일이 포함하고 있는 클래스의 정보를 표시하는데 위에 상수 풀에 인덱스를 가지고 있다. super_class는 이 클래스 파일의 슈퍼 클래스의 정보를 표시하는데 이 것 또한 상수 풀에 인덱스를 가지고 있다. interfaces_count, interfaces는 인터페이스 필드 내에 위치한 인터페이스 정보 개수, 실제 인터페이스 정보를 가진 상수 풀의 인덱스를 가진 배열의 형태이다. fields_count, fields는 클래스의 필드의 개수, 실제 필드의 정보를 나타내며 상수 풀의 인덱스를 가진다. methods_count, methods는 클래스의 메소드 개수, 메소드의 실제 정보를 나타내며 상수 풀의 인덱스를 가진다. attributes_count, attributes는 클래스의 속성의 개수, 속성에 대한 정보로 구성된다.

바이트코드는 자바 가상기계의 기계언어로 간주할 수 있고 자바 가상기계가 클래스 파일을 적재할 때 클래스 내에서 각각의 메소드에 대하여 스트림 형태로 얻어진다. 이때 스트림은 8 비트의 바이트로 구성된 이진 스트림 형태이다. 또한 바이트코드는 기본적으로 스택 지향 구조를 가지고 있고 처음부터 인터프리터를 목적으로 설계되었다. 즉, 자바 가상기계에 의해 인터프리트 되거나 클래스 파일 로딩시 컴파일된다. 한편, 바이트코드는 프로그램의 기능 추가 및 성능 향상을 위하여 바이트코드의 변경을 해야 하고 클래스 및 객체는 반드시 자바로 구성하여야 한다는 제약성도 가지고 있다.

2.2 .NET 환경의 PE File

.NET 환경에서 수행되는 마이크로소프트 제품군에 대한 .NET-지향 컴파일러를 통해 생성된 .NET 애플리케이션은 메타데이터와 IL 코드로 구성된 추

상화된 중간 표현이며 관리 모듈(managed module)이라 부른다. 메타데이터는 애플리케이션을 구성하는 각 요소(예; 클래스, 멤버, 속성 등)에 대한 정보와 외부 어셈블리 참조, 다른 애플리케이션과의 관계 등을 저장하고 있다. 이러한 메타데이터는 CLR의 로더에 의해 참조되며 IL 코드 수행 시에 참조될 수 있도록 내부 자료 구조로 저장된다. IL 코드는 MSIL 또는 CIL이라 부르며 실질적인 동작을 수행하는 부분으로서 바이너리 형식으로 엔코딩되어 있다. 이러한 IL 코드는 JIT 컴파일러에 의해 로더에 의해 생성된 정보를 참조하여 플랫폼에 기반을 둔 네이티브 코드로 생성된다[5,6].

.NET 실행 파일은 마이크로소프트 PE(Portable Executable)과 COFF(Common Object File Format)를 기반으로 하여 이를 확장한 형태로서 그림 2와 같이 CLR(Common Language Runtime) 환경에서 동작하기 위해 PE/COFF 헤더, CLR 헤더, CLR 데이터, 네이티브 데이터 및 코드로 구성되어 있다. .NET 실행 파일 형식은 기존의 실행 파일과 동일하게 확장자가 EXE 또는 DLL이며 스몰 엔디언(Small-Endian)의 순서로 저장된다. PE/COFF 헤더는 운영체제와 파일에 속성 정보 등을 저장하며 MS-DOS stub, PE signature 등으로 구성되어 있다[7].

CLR 헤더는 PE 파일이 .NET 실행 파일임을 나타내는 정보를 나타낸다. CLR 데이터는 프로그램이 어떻게 실행될지를 결정하는 메타데이터와 IL 코드로 구성된 관리 모듈로 구성되며 메타데이터는 데이터에 관한 데이터로써 타입 정의, 버전 정보, 외부 어셈블리 참조 그리고 다른 표준화된 정보로 이루어져 있고 IL 코드는 CLR 환경에서 실행되는 명령어들의 집합이다. Native Image부분은 imports/exports이 포함한 데이터와 코드 그리고 헤더들에 의해 묘사된 실질적인 데이터를 포함한다. 종류로는 data section, relocation section, text section, unmanaged resource section, thread local storage data section 등이 있다[8,9].

2.3 자바 가상기계의 로더/링커

클래스 파일을 입력으로 받은 자바 가상기계는 입력받은 클래스 파일에 대한 검증을 수행한 후 그림 1과 같은 과정을 거쳐 실행된다[10].

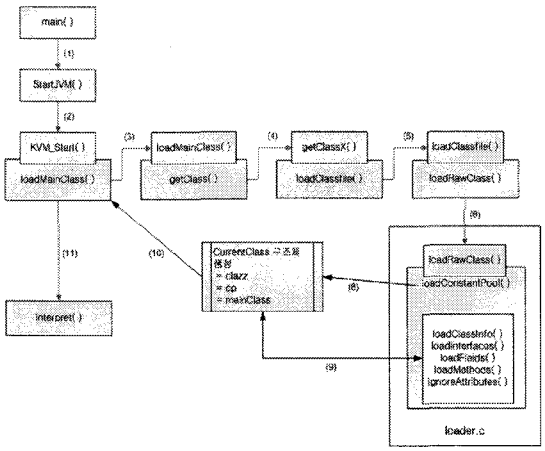

그림 1. 자바 가상기계 로더/링커

main() 함수가 실행되면서 시스템의 초기화 및 가상기계의 시작을 알리고 StartJVM() 함수를 호출하게 된다(그림 1-(1)). 호출된 StartJVM() 함수는 가상기계가 시작과 동시에 실제적 운용을 위해 KVM_Start() 함수를 호출하게 되며(그림 1-(2)) KVM_Start() 함수는 모든 것이 초기화되어 있고 실행가능한 상태의 가상기계를 호출하게 된다. 또한 loadMainClass() 함수를 통해 입력으로 받아들인 클래스 파일의 정보 추출 및 저장을 위한 준비를 한다(그림 1-(3)). loadMainClass() 함수는 getClass() 함수를 호출하면서 주어진 클래스 파일의 이름을 추출하고 필요하다면 클래스 파일을 로딩하기 시작한다. 이때 로딩을 위해 getClassX() 함수가 호출되며(그림 1-(4)) getClassX() 함수는 실제적으로 클래스를 로딩하기 위해 loadClassfile() 함수를 호출한다[5].

호출 된 loadClassfile() 함수는 시스템에서 주어진 클래스 파일을 로드 및 링크하기 시작한다. 이때 슈퍼인터페이스 계층 구조는 반복적으로 정보를 수집하며 인터페이스 계층 구조는 재귀적으로 정보를 추출한다. 슈퍼클래스 계층 구조의 정보를 수집하기 위해서 loadClassfile() 함수는 loadRawClass() 함수를 호출한다(그림 1-(5)). 호출된 loadRawClass() 함수는 loader.c 파일 안에 있으며 여기서 정보를 추출한다. 또한 클래스 파일에 있는 각종 정보를 추출하기 위해 loader.c 파일을 다시 호출한다(그림 1-(6)). loader.c의 실행을 통해 입력으로 들어 온 클래스 파일의 정보를 추출하며 추출된 정보들의 시작 주소값을 저장하기 위해 CurrentClass 구조체를 정의하여

저장한다(그림 1-(8),(9)). 모든 정보의 시작 주소값을 가지고 있는 CurrentClass 구조체는 후에 interpret함수에 의해 정보가 해석되어 진다(그림 1-(10),(11)). 이때 loader.c 파일 내의 Class loading operation 부분에서 호출되어지는 loadConstantPool함수, loadClassinfo함수, loadField함수등은 superclass를 비롯하여 최 상위 클래스의 정보까지 추출하게 된다.

추출한 정보는 메모리 내의 지정된 스택에 저장된다. 로더는 우선적으로 클래스 파일의 정보를 첫째로 StringPool 스택에는 Utf8정보가 저장되고, 둘째, RawPool 스택에는 Utf8과 태그정보외의 모든 정보가 저장된다. 마지막으로 Tag 스택은 각각의 상수풀 정보 앞의 태그값을 저장하게 된다. 이렇게 저장된 정보는 메모리 손실을 막기 위해 다시 한번 통합과정을 거쳐 저장된다. 즉, 기존에 사용했던 3개의 스택, StringPool 스택, RawPool 스택, Tag 스택들의 비어있는 공간을 제거하고 새로운 두개의 스택을 이용하여 다시 저장한다. 그 중 ConstantPool 스택에는 Utf8 정보를 저장했던 StringPool 스택을 제외한 나머지 스택에서 정보를 꺼내와 빈 공간 없이 저장을 하게 되며 이때 클래스 파일 내의 기타 정보들도 같이 저장된다. 그리고 CPTags 스택 또한 Utf8의 태그를 제외한 나머지 태그정보들이 저장된다. 이렇게 새로 생성된 새로운 2개의 스택을 이용하여 CurrentClass 구조체의 일부 정보를 구성하게 되고, 일부 완성 된 CurrentClass 구조체와 Utf8이 저장된 StringPool 스택을 이용하여 클래스 파일의 모든 정보를 추출 및 저장한다.

3. 실행 파일 포맷 설계 및 변환기

3.1 실행 파일 포맷 설계

본 논문에서 제시하는 실행 파일 포맷(Execution File Format; *.evm)은 임베디드 시스템에서 원활히 수행될 수 있도록 자바 클래스 파일 포맷을 기반으로 구조를 재설계하여 실행 속도 및 크기 개선에도 목적을 두었으며 .NET PE 파일의 정보를 충분히 저장할 수 있도록 고려하였다.

임베디드 시스템 등에서 사용할 목적으로 설계된 실행 파일 포맷은 여러 개의 클래스를 파일 안에 포함하며 8비트 스트림으로 구성되어 있다. 또한 비트 스트림은 리틀-엔디언으로 저장된다. 또한 EFF의 구조를 표현하는 방법으로 C의 구조체와 유사한 형태로 표현하며 타입 u1, u2, 그리고 u4는 unsigned 1-byte, 2-byte, 4-byte를 나타낸다.

```
EvmFile {
        u4 magic;
        u2 majorVersion;
        u2 minorVersion;
        u2 module;
        u2 language;
        u2 entrypoint;
    u1 VMCodesCount;
    VMCodeInfo VMCode[VMCodesCount];
        MetadataInfo Metatable[];
}
```

그림 2. EFF의 구조

magic 필드(0x0E054DFF)는 EFF을 식별하기 위한 엔트리이다. 실행 파일 포맷의 주 버전과 부 버전을 나타내는 엔트리이며 module 필드는 파일의 이름을 나타내는 엔트리이다. 이것은 MDTString index의 값을 가진다. language 필드는 소스 코드의 언어를 나타내는 엔트리이다. 이것은 MDTString index의 값을 가진다. 스트링의 정보는 java, C#이다. entryPoint 필드는 파일 안에 시작 메소드를 나타내는 엔트리이다. 이것은 MDTMethod index의 값을 가진다. VMCodeCount 필드는 코드 테이블의 총 개수를 나타낸다. VMCode[] 필드는 코드 테이블을 나타내며, 해당 메소드 정보와 EVM의 중간언어인 SIL이 저장된다. 마지막으로 Metatable[] 필드는 메타데이터의 정보를 나타내는 엔트리이며 MetadataInfo의 기본 구조를 가진다. 이 테이블의 개수는 태그의 존재에 따라 달라진다.

EFF 설계시에 자바 클래스 파일 포맷(CFF)을 기반으로 각각의 실행 파일 포맷 사이에서 같은 기능을 하는 것은 매핑하고 불필요한 부분은 삭제했다. 실제적으로 CFF를 기반으로 설계된 EFF는 Metadata Table의 구조체로서 총 15개의 구조체로 설계되어 있다. 이들 구조체들은 CFF와 EFF의 매핑을 통해 나오는 결과를 가지고 사용될 것이다. 참조되는 클래스 정보를 가지고 있는 MDTRefClass, 사용자 정의

클래스 정보를 가지고 있는 MDTDefClass, method 정보를 가지고 있는 MDTMethod, field 정보를 가지고 있는 MDTField, interface 정보를 가지고 있는 MDTInterface, string 정보를 가지고 있는 MDTString, 사용자 정의 string 정보를 가지고 있는 MDTUserString이 있으며, Integer 정보를 가지고 있는 MDTInteger, Float 정보를 가지고 있는 MDTFloat, Long 정보를 가지고 있는 MDTLong, Double 정보를 가지고 있는 MDTDouble, descriptor 정보를 가지고 있는 MDTDescriptor, 예외처리 정보를 가지고 있는 MDTException 등이 있다.

3.2 CFF에 대한 EFF 매핑

실제적으로 CFF를 EFF로 재설계하기 위해 매핑 정보에 따라 그림 3과 같은 매핑 관계를 정의하여 설계하였으며 매핑시 정보의 크기 및 내용이 변환되는 경우가 존재한다.. 이러한 매핑 구조는 중간언어를 만들기 위해 매우 중요한 작업이다.

EFF의 magic과 CFF의 magic은 1:1 매핑이 된다. 같은 4바이트 크기로서 0xCAFEBABE의 값이 0x0E054DFF로 바뀌게 된다. 마찬가지로 EFF의 majorVersion과 minorVersion은 CFF의 major_version, minor_version이 1:1 매핑이 되며 EFF의 module은 CFF의 상수풀 정보 일부와 attributes 정보 일부가 1:N 매핑이 된다. EFF의 language 정보는 새로 추가되는 부분으로 CFF와 매핑되는 정보가 없다. EFF의 entrypoint는 CFF의 super_class 일부 정보와 methods_count의 정보와 1:N 매핑이 되며 VMCodeCount는 attributes_count와 1:1매핑이 된다. EFF의 VMCode는 CFF의 methods 정보 일부와 attributes 정보 일부와 1:N 매핑된다. 마지막으로 EFF의 Metadata는 CFF의 constant_pool_count, constant_pool[], this_class, super_class, interfaces_count, interfaces[], fields_count, fields[], methods_count, methods[], attributes_count, attributes[] 정보의 일부와 1:N 매핑된다.

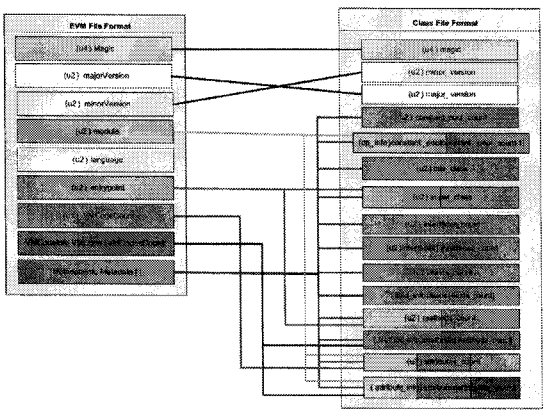

그림 3. CFF에 대한 EFF 매핑 관계

이러한 전체적인 매핑 관계를 바탕으로 하여 세부적인 매핑을 하게 되며, 그 중에서도 EFF의 Metadata Table과 CFF의 Constant Pool 정보의 매핑이 제일 중요하다. EFF와 CFF의 세부적인 매핑은 EFF의 Metadata[](module, entrypoint 포함)과 CFF의 magic, minor_version, major_version을 제외한 나머지 아이템들의 매핑을 의미한다. Metadata[]은 총 15개의 엔트리로 구성되어 있다.

3.3 실행 파일 포맷 변환

CFF는 내부적으로 총 16개의 요소가 있으며 각 요소의 설명과 요소들을 이용한 실제적인 구현 방법은 위에서 설명하였다. 다음 그림 4는 CFF의 16진수 코드로써 자바로 작성된 프로그램을 컴파일 했을 때 결과 값으로 나오는 클래스 파일의 내부 구조이다. 우선 한 줄에는 총 16바이트의 값이 나열되어 있으며, 1바이트 단위로 띄어쓰기가 되어있다. 즉 처음의 "CA FE BA BE"는 1바이트씩 계산되어 총 4바이트가 된다. 전체적인 구조를 보면 먼저 magic number가 4바이트 크기로 나오게 되고, minor와 major version이 2바이트씩 총 4바이트가 나온다. 그리고 Constant Pool의 개수를 나타내는 값이 2바이트, 이어서 Constant Pool 정보가 나오게 된다. 그 다음으로 access_flag, this_class 등의 정보가 나오며, 마지막에는 attribute 정보가 나오게 된다. 이렇게 여러 정보가 저장 된 클래스 파일은 JVM에 의해 해석되어 실행결과를 볼 수 있게 한다.

그림 3과 같은 매핑 관계에 따라 실제로 완성된 EFF 파일을 실제적으로 완성하는데 이를 위해 CFF의 16진수 코드를 분석하고 분석된 자료를 바탕으로 EFF의 실제적인 16진수 코드의 설계가 필요하다. 실제로 구현된 EFF는 내부적으로 그림 5와 같이 총 11개의 요소로 구성되어 있으며 총 16 바이트의 값이 1 바이트 단위로 나열되어 있다. 즉 처음의 "0E 05 4D FF"는 1바이트씩 계산되어 총 4바이트가 된다.

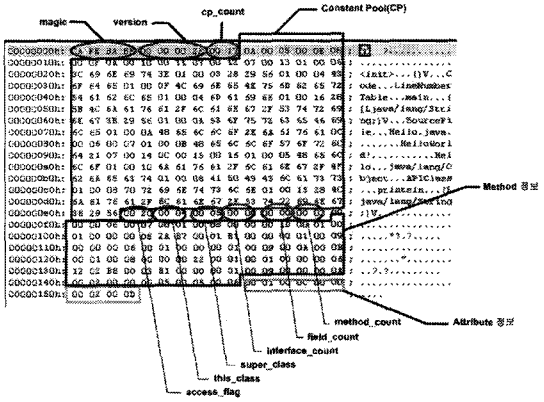

그림 4. CFF 16진수 코드

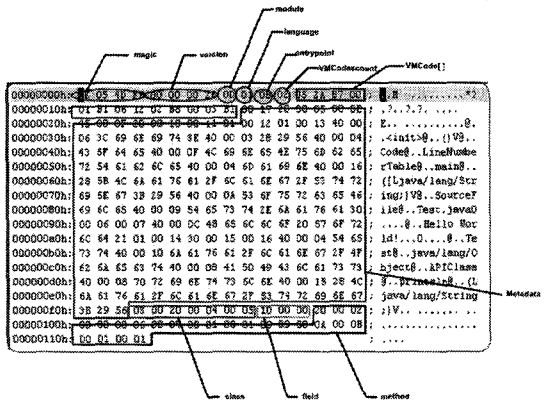

그림 5. EFF 16진수 코드

전체적인 구조를 보면 우선 4바이트 크기의 magic number와 version 정보가 나온다. 다음으로 1바이트 크기의 module, language, entrypoint, VMCodes-count 순으로 정보가 나오게 되며 가변적 크기를 가지고 있는 VMCode[]와 Metadata 정보가 이어서 나오게 된다. 마지막으로 class, field, method 순으로 정보가 나열된다.

4. 링커/로더 시스템

4.1 시스템 구성도

자바 클래스 파일과 .NET PE 파일을 기반으로 설계된 EFF(*.evm)을 실질적으로 실행하기 위한 가상기계의 내부 구조는 그림 6과 같이 크게 네부분으로 구성되어 있다. 본 논문에서 구현하고자 하는 로더/링커의 실행 구조는 그림 1과 같이 기존 가상

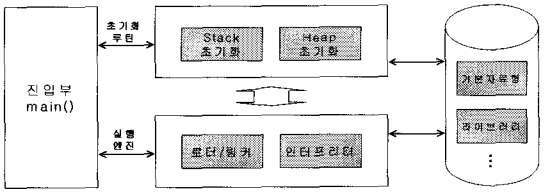

그림 6. 가상기계 내부 구조

기계의 로더/링커와 동일한 과정을 거친다. 하지만 기존 로더/링커는 클래스 파일 포맷을 입력으로 받아 처리하지만 본 논문에서는 EFF를 입력을 받아 실행에 적합한 메모리 포맷을 변환하는 로더/링커를 구현한다.

가상기계의 내부 구조에서 첫째, 진입부는 전체 가상기계의 main() 함수 역할을 한다. 즉, 가상기계 실행을 위해 Stack/Heap 공간의 초기화 루틴 호출과 가상기계의 실행 엔진 부분인 로더/링커, 인터프리터 호출과 같은 주요 동작을 수행한다. 둘째, 실질적으로 가상기계의 실행 엔진인 인터프리터와 로더/링커에 의해 사용되는 Stack/Heap 공간을 초기화 하는 루틴(VmInit())으로 구성되어 있다. 셋째, 주 함수로부터 호출되는 부분으로서 실행 파일 포맷을 메모리에 로딩하는 부분과 로딩된 정보를 실제로 실행하여 결과를 생성하는 인터프리터 부분으로 구성되어 있다. 마지막으로 기본 자료형 및 라이브러리 함수가 포함되어 있는 부분이다. 특히, 이 부분에서는 Windows와 Unix 환경을 위한 헤더가 정의되어 있으며 입력되어지는 실행 파일 포맷에서 원하는 정보를 바이트 단위로 얻기 위해 getXXX로 시작되는 매크로 함수가 정의되어 있다.

실질적으로 본 논문에서 설계한 실행 파일 포맷을 가상기계가 입력 받으면 main함수는 VmInit() 함수를 호출하여 로더가 추출한 정보를 저장하기 위해 만들어 놓은 스택 및 힙을 초기화 한다. 초기화 성공 여부를 main() 함수에 반환하고 초기화가 정상적으로 이루어지면 main() 함수는 VmStartApp() 함수를 호출하여 로더를 실행하게 된다. 로더가 실행되면 실행 파일 포맷을 읽어 들이고 위치 이동을 위해 선언되어진 포인터 변수에 처음 위치에 대한 주소값을 넘겨주게 된다. 이러한 과정을 거친 후 로더는 실행 파일 포맷 내부를 이동하면서 필요한 정보를 추출하게 되고 추출된 정보는 스택 또는 힙에 저장된다.

4.2 메모리 포맷

가상기계의 로더는 본 연구에서 설계한 실행 파일 포맷인 EFF(*.evm) 내부에서 필요한 여러 가지 정보를 추출한다. 추출한 정보는 그림 7과 같이 두개의 힙 공간(evmHashList, evmHeap)과 하나의 구조체(EvmStruct* evm)에 저장된다.

evmHashList는 전역 포인터 배열로 선언되어 있으며 처음 사용되는 구조체를 초기화한다. 즉, evmHeap에 저장되어 있는 클래스 정보의 시작 주소를 가지고 있다. evmHeap은 전역 포인터 변수로 선언되어 있으며 EFF의 모든 정보를 저장한다. 이때 저장되는 정보는 대부분 주소 값을 저장하여 나중에 EFF내에서 필요한 정보를 빠르게 찾을 수 있도록 한다. evm은 클래스의 정보가 저장되는 구조체로서 그림 8과 같은 구조를 가지고 있으며 새로운 클래스가 저장되면 반드시 저장되는 내용이다. 이 구조체를 통해 클래스의 기능과 위치를 알 수 있다.

로더는 EFF로부터 필요한 정보를 추출하여 evmHeap에 저장하는 순서는 그림 9와 같다.

먼저 main() 함수가 포함되어 있는 클래스의 분석 및 상수풀 내용이 evmHeap에 저장된후에 구조체의 변수중 attrib2 값을 검사하여 상위 클래스가 있는지

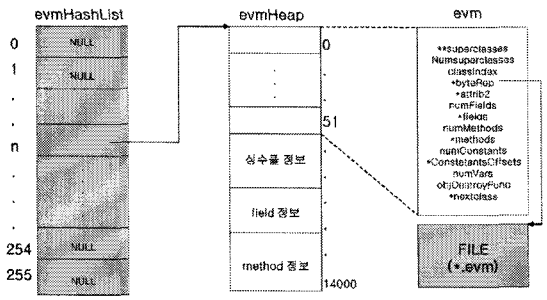

그림 7. 로더의 출력

구조체 \ 내용	변 수	저장 내용
EvmStruct	superclasses	상위클래스의 주소 저장
	numsuperclasses	상위클래스의 개수 저장
	byteRep	입력된 evm파일의 시작주소를 저장
	attrib2	상위 클래스가 있는지의 판별을 위한 주소 저장
	numFields	필드의 개수를 저장
	fields	필드정보가 저장되어 있는 주소 저장

그림 8. EEF의 구조체

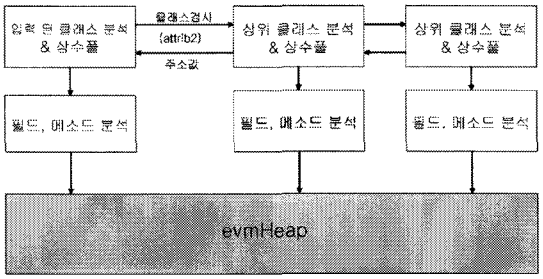

그림 9. EFF에 대한 메모리 적재 순서

를 확인한다. 상위 클래스가 존재한다면 상위 클래스의 분석 및 상수풀 내용을 evmHeap에 저장한다. 이와 같은 방법으로 최상위 클래스의 분석 및 상수풀의 내용까지 모두 저장하게 되면 최상위 클래스의 필드와 메소드 정보가 evmHeap에 저장된다. 이와 같은 과정을 반복한 후 main() 함수가 포함되어 있는 클래스의 필드와 메소드 정보를 저장하게 된다.

그림 10과 같은 자바 소스 프로그램에 대해 실질적으로 본 연구에서 구현한 로더를 통해 EFF 정보를 evmHeap에 저장하는 결과는 그림 11과 같다. 그림 10의 예제에서 main() 함수를 포함하는 Sub 클래스와 그 상위 클래스인 Super 클래스 그리고 Sub 클래스와 같은 위치인 friend 클래스를 작성하였으며 object 클래스는 최상위 클래스를 나타낸다.

4.3 실험 및 성능 평가

본 연구에서 구현한 EFF에 대한 가상기계의 로더/링커의 성능을 검사하기 위해 그림 12와 같이 화면 출력을 지원하는 네이티브 함수(APIClass.java)와 예제 프로그램을 이용하였다. Test.java에서는 기본적으로 실행되는 문자열을 화면에 출력하는 소스이

```
class Super{
        int super_;
        public static void for_loop(int i){
                for(int i=0; i>i; i++){
                        APIClass.println(i);
                }
}
public class Sub extends Super{
        int sub_;
        public static void main(String[] args){
                for_loop(5);
                friend.friends( );
        }
}
class friend extends Super{
        int friend_;
        public static void friends( ){
                for_loop(4);
        }
}
```

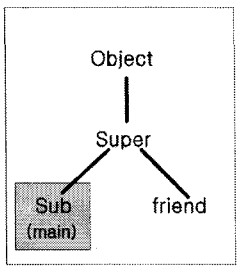

그림 10. 예제 프로그램 및 관계도

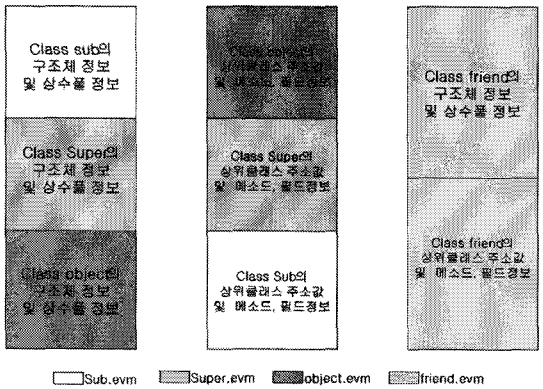

그림 11. evmHeap에 EFF의 저장 구조

```
//APIClass.java
public class APIClass {
   public static native void print(int value);
   public static native void println(int value);
   public static native void printsln(String value);
   public static native void prints(String value);
}

//Test.Java
class Test {
   public static void main(String[] args) {
        APIClass.printsln("Hello World!");
   }
}

//Plus.java
class Plus {
   public static void main(String args[]) {
        int i=1;
        int j=2;
        int c = i+j;
        APIClass.println(c);
   }
}
```

그림 12. 성능 측정을 위한 예제 프로그램

며, Plus.java는 간단한 덧셈을 하는 소스이다. 이 3개의 프로그램들은 본 연구에서 구현한 실행 파일 포맷 변환기를 통해 EFF로 변환하여 사용하였다.

Test.java의 결과는 다음 그림 13과 같이 VmInit() 함수가 호출되며 Test.evm의 내용과 함께 분석된 결과가 나오고 이어서 Object.evm, APIClass.evm의 분석 결과가 나온다. 모든 것이 분석되고 나면 결과가 출력된다.

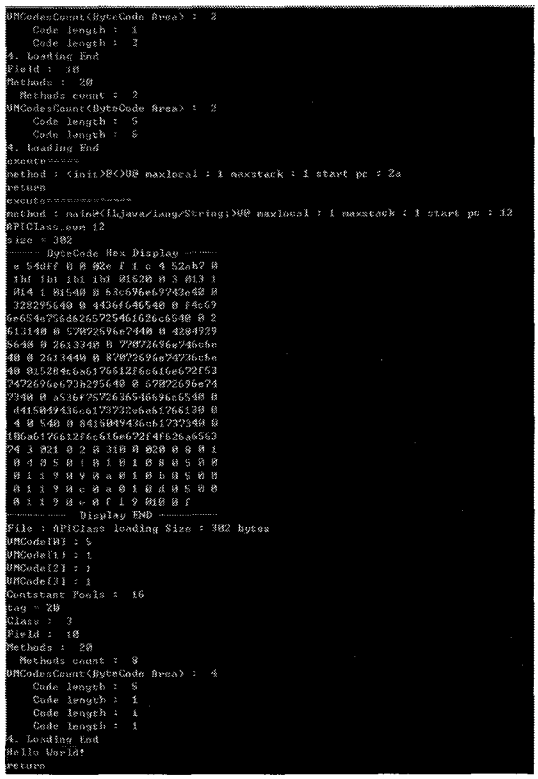

그림 13. Test.java 결과

본 연구에서 구현한 로더를 적용한 가상기계를 이용하여 EFF가 입력으로 주어졌을 때 나오는 결과를 이용하여 CFF(*.class)와 비교 분석을 하였다. 파일 크기 및 처리 속도 체크를 위한 환경으로 Microsoft 사의 Visual Studio에서 제공하는 코드 실행 시간 측정기를 사용하였다.

파일 크기의 비교는 표 1과 같이 Test.class와 Test.evm을 비교하여 Test.class는 파일크기가 340 바이트, Test.evm 파일은 279바이트로 61바이트 차이가 난다. Plus.class와 Plus.evm의 파일 크기를 비교해도 73바이트 차이가 난다. 이렇게 파일의 크기가 축소됨에 따라 파일을 읽어 들이는 속도를 개선할 수 있었다.

표 1. 파일 크기 비교(단위 : byte)

파일이름	*.class	*.evm
Test	340	279
Calculation	322	249
Simple	499	386

처리 속도는 프로그램을 작동하는 CPU와 운영체제에 따라 변동될 수 있지만 본 연구에서의 처리 속도의 비교를 위해 CPU(펜티엄3 1G 코퍼마인), 운영체제(Win. XP)를 이용하여 표 2와 같은 결과를 얻었다.

아래와 같은 실험 결과를 통해 실행 파일 포맷 크기, getevm() 함수의 처리 횟수, 로더에 의해 추출된 자료가 저장되는 자료구조 개선에 따른 차이로 인해 파일의 크기 및 처리 속도에서 우수한 성능을 확인하였다.

표 2. 처리 속도 비교(단위 : ms)

	*.class	*.evm
Test.java	66.344	0.872
Calculation	57.241	0.359
Simple	85.442	1.021

5. 결 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 활성화되고 있다. 특히, 자바 언어 작성된 프로그램을 플랫폼에 의존하지 않고 실행할 수 있는 환경을 지원하는 자바 가상기계와 마이크로소프트사에서 심혈을 기울이고 있는 .NET 실행시간 환경 기술은 현재 전 세계적으로 응용 분야가 확대되고 있다.

로더/링커는 자바 클래스 파일, .NET 환경의 중간 표현인 PE 파일을 각각 입력으로 받아 이를 실행 환경으로 적재하는 로딩 및 검증, 준비, 레졸루션 동작을 수행하는 링킹 동작을 수행하며 원활한 수행을 위한 초기화, 실행에 필요한 최적화된 정보 저장 등 실행에 필요한 모든 정보 생성 및 무결성을 보장하는 아주 중요한 부분이다.

본 논문에서는 자바 클래스 파일과 .NET 환경의 PE 파일에 대한 통합 로더/링커 시스템을 설계하고 구현한다. 이를 위해, 자바 클래스 파일과 .NET PE 파일 정보를 모두 저장할 수 있는 새로운 실행 파일 포맷(*.evm) 및 메모리 포맷을 설계했으며 저장된 실행 정보를 활용하여 JVM 또는 .NET 환경에서 실행할 수 있도록 링커/로더 시스템을 구현하였다.

또한 본 연구에서 제안한 새로운 실행 파일 포맷(*.evm)을 활용하여 기존의 자바 클래스 파일과 크기, 실행 속도 등에 대한 비교 실험을 통해 소규모 가상기계의 개발시에 활용될 수 있는 기반을 구축하였다. 향후에 .NET PE 파일과의 비교를 통해 보다 다양한 분야에서 효과적을 사용될 수 있도록 한다. 또한 통합/로더 링커 구현에서 내부 자료 구조를 개선하여 로더/링커의 자체의 실행 속도 및 효율성을 개선하고자 한다.

참 고 문 헌

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

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

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

[4] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM TOPLAS*, Vol.11, No.4, pp. 491-516, Oct. 1989.

[5] Don Box and Chris Sells, *Essential .NET Volume 1 The Common Language Runtime*, Addison Wesley, 2002.

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

[7] John Gough, *Compiling for the .NET Common Language Runtime(CLR)*, Prentice-Hall, 2002.

[8] James S. Miller, *The Common Language Infrastructure Annotated Standard*, Addison Wesley, 2003.

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

[10] Bill Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 2000.

고 광 만

1991년 2월 원광대학교 컴퓨터공학과(공학사)
1993년 2월 동국대학교 컴퓨터공학과(공학석사)
1998년 2월 동국대학교 컴퓨터공학과(공학박사)
2001년 8월 광주여자대학교 컴퓨터과학과(전임강사)
2002년~2003년 Queensland University of Technology 연구교수
2001년~현재 상지대학교 컴퓨터정보공학부 부교수
관심분야 : 프로그래밍언어론 및 컴파일러