simpleRTJ 임베디드 자바가상기계의 ROMizer 분석 연구

양 희 재†

요 약

전용 목적의 임베디드 자바 시스템은 일반적으로 프로그램을 이루는 모든 클래스 파일들이 호스트 컴퓨터 상의 ROMizer에 의해 하나의 ROM 이미지로 변형되고, ROM에 적재된 이미지를 임베디드 시스템의 자바가상기계가 실행하는 모델을 따른다. 제한된 크기의 메모리 자원과 낮은 성능의 프로세서를 갖는 임베디드 시스템에서는 이 ROM 이미지를 어떤 형식으로 정의하는가 하는 것이 매우 중요하다. 그 형식에 따라 원래의 클래스 파일보다 훨씬 적은 크기의 메모리 만으로도 전체 클래스 정보를 담을 수 있게 되며, 또한 클래스 내부 정보에 신속하게 접근할 수 있게 되기 때문이다. 본 논문에서는 simpleRTJ라고하는 임베디드 자바가상기계에서 구현된 ROMizer와, 특히 그것이 생성하는 ROM 이미지의 형식에 대해 분석해보았다. 분석 결과 ROMizer는 원래 클래스 파일에 비해 절반 이상의 메모리 절감 효과를 얻게 하며, 최대 6배 이상의 속도로 클래스 내부 정보를 접근할 수 있게 하는 것으로 밝혀졌다. 이 연구의 결과는 ROM 기반의 임베디드 자바 시스템을 위한 보다 효율적인 ROMizer 의 개발에 적용되어질 수 있을 것으로 기대된다.

Analysis of the ROMizer of simpleRTJ Embedded Java Virtual Machine

Heejae Yang†

ABSTRACT

Dedicated-purpose embedded Java system usually takes such model that all class files are converted into a single ROM image by the ROMizer in the host computer, and then the Java virtual machine in the embedded system executes the image. Defining the ROM image is a very important issue for embedded system with limited memory resource and low-performance processor since the format directly influences on the memory usage and effectiveness of accessing entries in classes. In this paper we have analyzed the ROMizer and especially the format of the ROM image implemented in the simpleRTJ embedded Java virtual machine. The analysis says that memory space can be saved up to 50% compared to the original class file and access speed exceeds up to six times with the use of the ROMizer. The result of this study will be applied to develop a more efficient ROMizer for a ROM-based embedded Java system.

키워드 : 임베디드 시스템(Embedded System), 자바(Java), 자바가상기계(Java Virtual Machine), 클래스 파일(Class File)

1. 서 론

한번 프로그램을 작성하면 어느 플랫폼에서나 그 프로그램을 사용할 수 있다는(*Write Once, Run Anywhere*) 자바의 플랫폼 독립성은 다양한 하드웨어 및 운영체제를 갖는 임베디드 시스템을 위한 좋은 대안이 될 수 있다는 점에서 최근 더욱 큰 관심을 받고 있다[1].

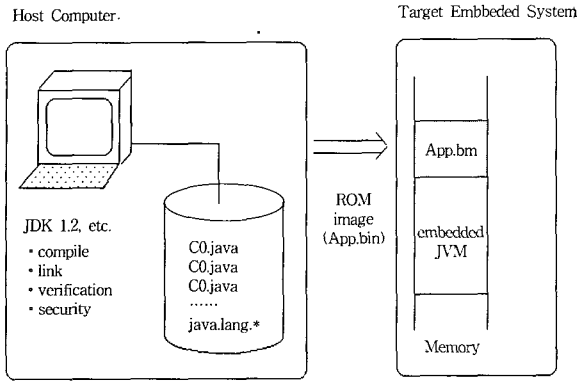
자바 프로그램의 실행을 위해서는 해당 플랫폼에 자바가상기계(Java Virtual Machine)가 설치되어야 한다. 모든 자바 프로그램은 JVM 상에서 실행되어지며 임베디드 시스템도 예외가 아니다. 임베디드 시스템을 위한 대표적인 JVM으로서는 국외의 경우 KVM, PERC, ChaiVM, Jbed, LeJOS, simpleRTJ 등이 있으며, 국내에서도 TeaPot, SK-VM, ez-Java 등의 제품들이 사용 중이다.

JVM 상에서 실행되어지는 자바 프로그램은 다수개의 클래스들로 이루어진다. 클래스에 대한 모든 정보는 클래스 파일에 저장되며, 프로그램 실행시 필요한 클래스 파일들이 지역 디스크 또는 네크워크를 통해 동적으로 적재되어진다.

데스크톱 방식의 컴퓨터에서는 필요한 클래스 파일들이 지역 디스크 상에 놓여져 있는 것이 일반적이다. 그러나 임

※ 이 논문은 2001학년도 경성대학교 연구년 연구비에 의해 지원되었음.
† 정 회 원 : 경성대학교 컴퓨터공학과 교수
논문접수 : 2003년 7월 19일, 심사완료 : 2003년 9월 18일

베디드 시스템에서는 디스크와 같은 보조기억장치가 없는 경우가 대부분이므로 필요한 클래스 파일들은 별도의 호스트 컴퓨터에서 개발된 후 임베디드 시스템의 ROM에 적재되거나 또는 네트워크를 통해 다운로드 된다(그림 1).



(그림 1) ROM 기반 임베디드 자바 시스템 환경

ROM에 클래스 파일을 둘 때는 메모리 사용량을 줄이고 접근 속도를 빠르게 하기 위해 원래의 클래스 파일 규격과 다른 변형된 형태로 둔다. 이와 같이 다수개의 클래스 파일을 변형시켜 하나의 ROM image화 시켜주는 프로그램을 *ROMizer*라고 부른다 [2]. PERC, LeJOS, simpleRTJ 등 다수의 임베디드 JVM들에서 ROMizer 기능이 사용되고 있다.

본 연구의 목적은 이와 같은 ROMizer에 대한 분석, 특히 그것이 만들어내는 ROM 이미지 형식의 성능에 대해 조사해보고자 하는 것이다. 여기서 성능이란 원래의 클래스 파일 크기에 비해 이미지 크기는 얼마나 줄어들었는지, 그리고 클래스를 이루는 개별 항목에 대해 얼마나 신속히 접근이 이루어질 수 있는지에 대한 내용으로 나눌 수 있다. 다수의 상용 제품에 ROMizer가 사용되고 있지만 그것의 구조를 밝히는 논문이나 또는 ROM 이미지 형식의 성능을 다룬 논문은 아직까지 찾아보기 어렵다. 대개의 경우 상용 제품으로서 그 원천코드가 제공되고 있지 않고 있기 때문이다.

본 논문에서는 simpleRTJ에서 제공되는 ROMizer에 대해 분석해 보았다. simpleRTJ[3]는 RTJ Computing 사에서 개발한 8/16/32비트 프로세서용 임베디드 자바 시스템으로서, ROMizer 자체의 원천코드는 제공되고 있지 않지만 상용 제품임에도 불구하고 JVM의 원천코드는 공개되어있다. 본 연구에서 우리는 simpleRTJ에서 채택한 ROMizer의 동작에 대해 조사해보고, 그것이 만들어내는 ROM 이미지의 형식을 분석하여 그 성능을 평가해 보았다. 실험 결과 simpleRTJ의 ROMizer가 생성하는 ROM 이미지는 원래 클래스 파일에 비해 최대 6배까지 접근 속도의 향상이 가능했으며, 클래스 파일 크기에 비해 절반 정도의 메모리 절감 효과가 있었다. 본 연구의 결과는 추후 ROM에 기반한 임베디드 자바 시스템을 위한 보다 효율적인 ROMizer의 개발에 적용되어질 수 있을 것으로 기대된다.

본 논문의 구성은 다음과 같다. 2장에서는 클래스 파일의 대략적 구조 및 레졸루션에 대해 설명하고, ROMizer가 담당해야되는 프리레졸루션에 대해 소개한다. 3장에서는 simpleRTJ ROMizer의 기능 및 그것이 만들어내는 ROM 이미지에 대한 전체 모습을 나타내었으며, 4장에서 ROM 내에서 클래스들이 어떻게 배치되는지, 그리고 클래스를 이루는 필드, 메소드, 상수 등이 어떻게 표현되며 접근되는지에 대해 설명한다. 5장에서 simpleRTJ ROMizer가 만들어 낸 ROM 이미지의 성능을 정량적으로 분석해 보며, 6장에서 결론을 맺는다.

2 배경 연구

2.1 클래스 파일과 레졸루션

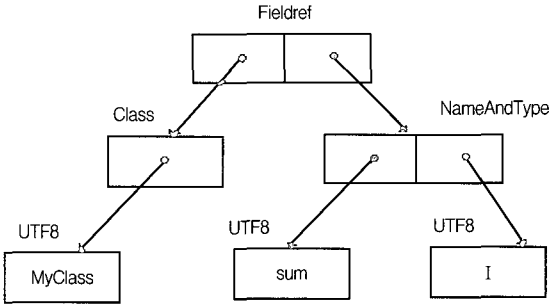
앞서 말한 바와 같이 하나의 자바 프로그램은 다수개의 클래스들로 구성되며, 각 클래스들은 각기 해당 클래스 파일에 저장된다. 즉 하나의 클래스 파일은 하나의 클래스에 대한 모든 정보를 갖는다.

클래스 파일의 구조는 자바가상기계 명세에서 정한 바에 따라 헤더, 상수 풀(constant pool), 클래스 정보, 필드 정보, 메소드 정보 등 모두 다섯 개 부분으로 구성된다[4]. 헤더는 매직 번호와 버전 번호로 이루어지는 8바이트의 고정된 영역이며, 상수 풀에 대해서는 아래에 따로 설명한다. 클래스 정보는 이 클래스의 접근 제어값, 자신 및 상위 클래스의 이름, 인터페이스의 이름 등을 나타내고, 필드 정보와 메소드 정보는 각각 이 클래스가 갖는 필드와 메소드에 대한 정보를 가지고 있다.

클래스 파일에서 가장 큰 크기를 차지하는 부분은 상수 풀이다. 상수 풀은 클래스에서 사용되는 모든 상수들을 모아 둔 곳으로서 한 연구에 따르면 이것의 크기는 전체 파일 크기의 60퍼센트 이상을 차지하는 것으로 알려져 있다 [5]. 상수 풀은 연산의 오퍼랜드로 사용되는 일반 상수 뿐 아니라 클래스나 필드, 그리고 메소드 등의 이름과 형식 등을 나타내는 상수들도 함께 포함하고 있다. 이들 상수들은 모두 상수 풀 인덱스 값에 의해 간접 접근된다. 이 상수들은 문자열로 이루어진 기호명으로 구성되어 있으며 실행시 이 기호명들을 사용하여 실제 엔트리 간의 연결이 이루어지게 되는데, 이런 연결 과정을 레졸루션(resolution)이라고 한다.

(그림 2)는 필드에 대한 레졸루션의 예를 보여준다. 필드에 대한 접근은 getfield #i와 같은 바이트코드 명령에 의해 이루어지는데, 여기서 i는 상수 풀의 해당 인덱스 값이다. 상수 풀의 i 번째 항목은 Fieldref 형식임을 알 수 있으며,

Fieldref 항목은 다시 Class와 NameAndType 항목을 가리킨다. Class 항목은 다시 UTF8 항목을 가리키며, 여기서 우리는 이 필드가 속한 클래스명이 MyClass 임을 알 수 있게 된다. 또한 NameAndType 항목은 다시 두 개의 UTF8 항목을 가리키는데, 이 항목들로부터 우리는 이 필드의 이름이 sum이며, 그것의 형식이 정수형(I)이라는 것을 알 수 있게 된다. 따라서 이 레졸루션이 끝날 때까지는 총 6번의 상수 풀에 대한 접근이 필요하다.



(그림 2) 필드에 대한 레졸루션 과정

이 예제에서와 같이 레졸루션은 많은 부담을 필요로 하는 작업이다. 실제로는 더 많은 부담이 필요하다. 즉 상수 풀의 각 항목들은 그 크기가 가변적이기 때문에 인덱스 값을 주었다고 해서 바로 해당 항목을 찾을 수는 없으며, 일반적으로는 별도의 인덱스 테이블을 메모리 상에 만들고 이 인덱스 테이블이 다시 상수 풀의 해당 항목을 가리키도록 구현한다. 따라서 필드에 대한 레졸루션을 위해서는 위에서 말한 6번의 인덱스 테이블에 대한 접근과 6번의 상수 풀에 대한 접근 등 모두 12번의 메모리 접근이 필요하다.

메소드에 대한 레졸루션도 동일한 과정을 밟으며 총 12번의 메모리 접근이 소용된다. 오퍼랜드로 사용되는 일반 상수에 대해서는 1번의 인덱스 테이블 접근 및 1번의 상수 풀 접근 등 2번의 메모리 접근이 필요하며, 클래스에 대해서는 2번의 인덱스 테이블 접근과 2번의 상수 풀 접근 등 4번의 메모리 접근이 필요하다. <표 1>은 클래스, 상수, 필드, 메소드 등 네 가지 항목의 레졸루션에 따른 메모리 접근 횟수를 정리한 것이다.

〈표 1〉 레졸루션에 따른 메모리 접근 횟수

	인덱스 테이블 접근 횟수 (A)	상수 풀 접근 횟수 (B)	전체 메모리 접근 회수 (A + B)
클래스	2	2	4
상 수	1	1	2
필 드	6	6	12
메소드	6	6	12

2.2 ROMizer 및 프리레졸루션

ROMizer는 다수개의 클래스 파일을 읽어들여 하나의 ROM 이미지를 만들어낸다. 필요한 클래스들은 프로그램 실행 전에 미리 정해지므로 이 환경에서는 동적 클래스 적재가 사용되지 않는다. 따라서 일반적 경우와 달리 실행 시간 전에 ROMizer에 의해 레졸루션이 이루어지게 된다. 이와 같이 실행 시간전 미리 이루어지는 레졸루션을 프리레졸루션(pre-resolution)이라고 하며, ROMizer의 중요 기능 중 하나이다 [2, 6].

프리레졸루션은 상수 풀에 들어있는 모든 기호명의 참조값들을 인덱스 또는 포인터 값으로 변환시켜준다. 이렇게 함으로서 상수 풀의 대부분을 차지하는 기호명들이 사라지게 되어 메모리의 절감을 얻을 수 있고 또한 필드 등 특정 항목에 대해 신속히 접근할 수 있게 된다. 본 논문에서는 simpleRTJ에서 채택하고 있는 프리레졸루션에 대해 조사해 보고, 프리레졸루션 결과 어느 정도의 메모리가 절감되었으며 특정 항목을 접근하는데 따른 메모리 접근 횟수는 얼마인지에 대해 분석하였다.

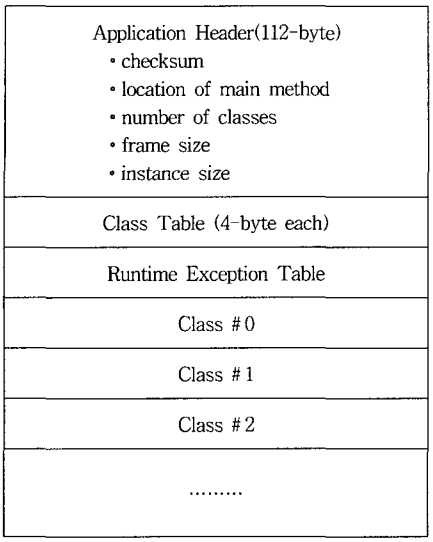
3 simpleRTJ ROMizer

simpleRTJ의 ROMizer는 ClassLinker라는 이름을 가지며 자바 언어로 작성된 것이다. 원천코드는 제공되지 않지만 ROM 이미지를 사용하는 JVM의 원천코드가 공개되어있으므로 ClassLinker의 구조 및 기능을 분석할 수 있었다. 본 논문에서는 ClassLinker 1.2.1 버전을 대상으로 했으며, Linear 16MB 메모리 모델을 사용했다.

3.1 ROM 이미지 개요

(그림 3)은 ClassLinker가 만들어 낸 ROM 이미지의 구성을 나타낸 것이다. 처음 112바이트는 헤더 부분으로서 오류검출을 위한 checksum, 프로그램 시작점에 해당되는 main 메소드의 위치, 이 프로그램에서 사용되는 클래스의 개수, 그리고 자바 스택 프레임의 크기와 각 인스턴스의 크기 등 정보로 이루어진다. 특기할만한 사항은 simpleRTJ에서는 모든 자바 스택 프레임의 크기와 모든 인스턴스의 크기를 각각 동일한 값으로 통일했다는 것이다[7]. 즉 자바에서는 어떤 메소드가 호출될 때마다 그 메소드 실행을 위한 별도의 오퍼랜드 스택과 지역 변수 배열로 이루어지는 자바 스택 프레임이 만들어지는데, simpleRTJ에서는 모든 메소드에 대해 동일한 크기(즉 최대 크기)의 메모리를 할당했다. 또한 새로운 인스턴스가 생성될 때마다 그 인스턴스를 위한 메모리가 할당되는데, simpleRTJ에서는 인스턴스의 종류에 관계없이 모두 동일한 크기(즉 최대 크기)의 메모리를 할당했다. 이같이 함으로서 자바가상기계에서 중요한 메모리 할

당 및 쓰레기 수집기의 구현을 쉽게 하고 있다 [8]. 이것은 뒤에 설명할 필드 및 메소드 정보의 구성에도 영향을 준다.

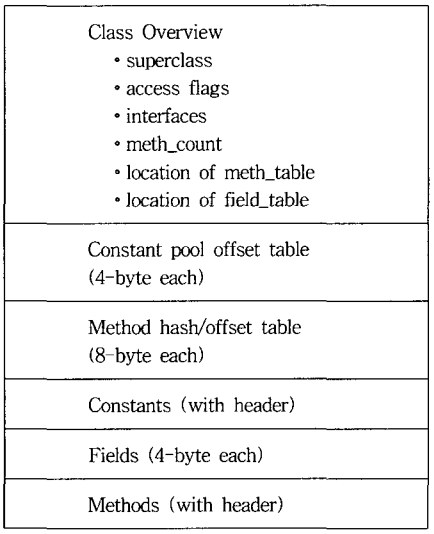


(그림 3) simpleRTJ ROM 이미지의 구성

헤더 다음에는 클래스 테이블이 이어지는데, 이것은 특정 클래스가 ROM 이미지 상의 어느 위치에 놓여있는지를 알려주는 32비트 포인터들의 집합이다. 다음으로는 런타임시 일어날 수 있는 11개의 예외/오류 클래스들의 위치를 알려주는 예외 클래스 테이블이 놓인다.

3.2 개별 클래스 개요

예외 클래스 테이블에 이어 실제 개별 클래스들이 놓인다. 클래스들은 각각 다른 크기를 가지며, 내용은 클래스 파일의 그것과 같지만 형식은 매우 다르다. 가장 중요한 차이는 ROMizer, 즉 ClassLinker에 의해 프리레졸루션이 이루어진 상태이므로 상수 풀에 있었던 기호명으로 된 참조값들이 없다는 것이다. 이들 기호명 참조값들은 모두 인덱스 값이나 포인터 값으로 변환되어 각 엔트리 구조체에 포함된다. (그림 4)는 ROM 이미지 속에 위치한 개별 클래스 내용을 보인 것이다.



(그림 4) 개별 클래스의 내용

이 그림에서 알 수 있듯이 ROM 이미지 내에서 각 클래스는 클래스 정보, 상수 풀 오프셋 테이블, 메소드 오프셋 테이블, 상수 정보, 필드 정보, 메소드 정보 등 여섯 부분으로 나뉜다. 여기서 상수 풀 오프셋 테이블은 2.1절에서 언급한 바와 같이 클래스 파일의 상수 풀에 대한 접근을 가능하게 하는 인덱스 테이블의 역할을 하며, 메소드 오프셋 테이블은 이 클래스가 가지고 있는 모든 메소드에 대한 개별 위치를 알려주는 포인터들의 모음이다. 가장 중요한 클래스 정보, 상수 정보, 필드 정보, 메소드 정보 등에 대해서는 다음 장에서 자세히 살펴보도록 하자.

4 클래스 구성

이 장에서는 ClassLinker가 만들어 낸 ROM 이미지 속에 들어있는 개별 클래스의 구조에 대해 상세히 분석해본다 (그림 4). 다음은 simpleRTJ ROM 이미지 및 JVM 원천코드를 분석한 결과이다.

4.1 클래스 정보

클래스 정보는 원래 클래스 파일에 들어있는 내용을 모두 포함하고 있으며, 다음과 같은 구조체로 정의되어 있었다.

```
struct class_T {
  uint16      flags ;        /* class modifier flags */
  uint16      index ;
  uint8       *ifaces ;
  uint8       *super ;       /* pointer to super class */
  uint8       *run ;         /* pointer to run() method, if
                                exists */
  uint8       *clinit ;      /* pointer to <clinit> method */
  uint32      meth_count ;
  uint8       *meth_tbl ;    /* pointer to hash/meth offsets tbl */
  uint8       *field_tbl ;   /* pointer to fields table */
} ;
```

여기서 인터페이스나 상위 클래스를 가리키는 부분이 클래스 파일에서와 달리 인덱스 값이 아니라 포인터 값으로 되어있음을 알 수 있다. 즉 ROMizer에 의해 프리레졸루션이 일어난 것이다. 또한 자기 클래스의 이름을 나타내는 기호명 대신 index 숫자가 들어가 있는 것을 볼 수 있다. 이것 역시 프리레졸루션의 결과이다. 이 구조체에서 필드의 개수를 알려주는 요소가 없는 까닭은 앞서 설명한 바와 같이 simpleRTJ에서는 모든 인스턴스의 크기를 동일한 것으

로 가정했기 때문이다.

class_T 구조체는 새로운 객체를 만드는 바이트코드 명령 new에 의해 참조된다. 이 명령의 형식은 new #i와 같으며, 클래스 파일에서 i는 상수 풀의 인덱스 값이지만 ROMizer에 의해 i 값은 (그림 4)에서 보인 상수 풀 오프셋 테이블의 인덱스 값으로 변환된다. 따라서 new 명령에 따른 메모리 참조 횟수는 ① 인덱스 값으로 상수 풀 오프셋 테이블 읽기, ② 테이블이 가리키는 주소의 class_T 구조체 읽기 등 2번이다.

4.2 상수 정보

여기서 말하는 상수는 바이트코드 연산 실행시 실제로 오퍼랜드로 사용되어지는 숫자나 문자를 의미한다. 즉 레졸루션 등에 사용되는 기호명 등은 이미 ROMizer에 의해 처리되었기 때문에 해당 사항이 없다. simpleRTJ에서는 다음 구조체로 상수 정보를 정의하고 있었다.

```
struct const_T {
 uint32 type_len ;
} ;
```

이 구조체는 32비트 크기를 갖지만, 실제로는 하위 16비트만이 이용된다. 이 16비트 중 상위 2비트는 상수의 형식을 나타내며, 나머지 14비트는 길이를 나타낸다. 형식은 세 가지가 있으며, int는 1, float는 2, 그리고 문자열은 3이다. simpleRTJ에서 double 및 long 형식은 지원되지 않는다.

이 구조체에 이어 실제 값이 놓인다. 예를들어 정수 값 50,000인 경우 상수 정보는 16진수로 00004004 0000C350 와 같이 되며, 문자열 "Hello"인 경우에는 이 문자열의 16비트 해쉬값인 0C53이 포함되어 0000C005 0C53 48656C6C6F가 된다.

const_T 구조체는 상수값을 오퍼랜드 스택 상에 적재시키는 바이트코드 명령 ldc에 의해 참조된다. 이 명령의 형식은 ldc #i와 같으며, 클래스 파일에서 i는 상수 풀의 인덱스 값이지만 ROMizer에 의해 i 값은 (그림 4)에서 보인 상수 풀 오프셋 테이블의 인덱스 값으로 변환된다. 따라서 ldc 명령에 따른 메모리 참조 횟수는 ① 인덱스 값으로 상수 풀 오프셋 테이블 읽기, ② 테이블이 가리키는 주소의 const_T 구조체 및 실제 값 읽기 등 2번이다.

4.3 필드 정보

필드 정보는 다음 구조체로 정의되었다.

```
struct field_T {
 uint32 type_index ;
} ;
```

이 구조체는 32비트 크기를 갖지만, 실제로는 그 중 하위 16비트만이 이용된다. 이 16비트 중 상위 4비트는 필드의 형식을 나타내며, byte, char, short, int, float, boolean, class, array 가 있고 각각의 값은 0부터 7에 해당된다.

나머지 12비트는 그 필드가 인스턴스 변수 배열의 몇 번째에 해당하는지를 나타내는 인덱스 값이다. 어떤 클래스의 인스턴스가 만들어지면 그 인스턴스를 위해 인스턴스 변수, 즉 가상 필드들을 저장할 수 있는 공간이 배열 형식으로 만들어진다. 각 배열 엔트리는 32비트의 크기를 가진다. 이 배열은 그 클래스가 갖는 필드 뿐 아니라 클래스 계층 상에 놓인 모든 상위 클래스들의 필드들을 수용할 수 있는 크기를 갖는다. 가장 상위 클래스가 갖는 필드들에게 인덱스 번호 0, 1, 2, …, n-1 까지가 할당되고, 그 다음 하위 클래스의 필드에게는 n, n+1, n+2, …, m-1 등과 같이 할당된다($n \leq m$).

원래의 클래스 파일에 저장되어 있는 필드 정보와 비교하면 ROMizer에 의한 프리레졸루션 결과 필드가 속한 클래스 위치는 상수 풀 오프셋 테이블 내에 흡수되었고, 필드의 이름과 형식은 미리 확인되어 ROM 이미지 상에는 나타나지 않는다.

field_T 구조체는 필드 값을 참조하는 바이트코드 명령 getfield/putfield 등에 의해 참조된다. 이 명령의 형식은 getfeld #i 등과 같으며, 클래스 파일에서 i는 상수 풀의 인덱스 값이지만 ROMizer에 의해 i 값은 (그림 4)에서 보인 상수 풀 오프셋 테이블의 인덱스 값으로 변환된다. 따라서 getfield/putfield 명령에 따른 메모리 참조 횟수는 ① 인덱스 값으로 상수 풀 오프셋 테이블 읽기, ② 테이블이 가리키는 주소의 field_T 구조체 읽기 등 2번이다.

4.4 메소드 정보

마지막으로 메소드 정보를 나타내는 구조체에 대해 살펴보자.

```
struct method_T {
 class_t *class_ptr ;
 uint16      flags ;          /* method flags */
 uint16      locals ;         /* size of locals */
 uint16      hash ;
 uint16      nargs ;          /* number of arguments */
 uint16      blen_idx ;       /* bytecodes length */
 uint16      unused ;
} ;
```

이 구조체의 항목들 중 flags, locals, blen_idx 등의 값은 클래스 파일 내에 포함되어있는 내용과 동일하다. 이 메소드가 실행될 때 필요한 스택의 크기는 클래스 파일 내에는 포함되어 있지만 이 구조체에서는 생략되어있는데, simpleRTJ

는 모든 메소드들에 대해서 동일한 크기의 스택. 즉 가장 많은 스택 공간을 요구하는 메소드와 동일하게 스택 공간을 할당하기 때문에 별도의 스택 크기 정보는 포함되지 않는 것이다.

원래의 클래스 파일에 저장되어 있는 메소드 정보와 비교하면 ROMizer에 의한 프리레졸루션 결과 이 메소드가 속한 클래스 위치는 상수 풀 오프셋 테이블 내에 흡수되었고, 메소드의 이름과 형식은 미리 확인되어 ROM 이미지 상에는 나타나지 않게 된다.

method_T 구조체는 메소드를 호출하는 바이트코드 명령 invokevirtual 등에 의해 참조된다. 이 명령의 형식은 invokevirtual #i 등과 같으며, 클래스 파일에서 i는 상수 풀의 인덱스 값이지만 ROMizer에 의해 i 값은 (그림 4)에서 보인 상수 풀 오프셋 테이블의 인덱스 값으로 변환된다. 따라서 invokevirtual 등 명령에 따른 메모리 참조 횟수는 ① 인덱스 값으로 상수 풀 오프셋 테이블 읽기 ; ② 테이블이 가리키는 주소의 method_T 구조체 읽기 등 2번이다.

5 평가 및 분석

이 장에서는 ClassLinker ROMizer가 생성한 ROM 이미지의 성능에 대해 평가해보았다. 여기서 성능이란 자바가상기계가 ROM 이미지 내에 포함되어 있는 클래스, 상수, 필드, 메소드 등의 항목에 대해 얼마나 적은 메모리 접근만으로 도달할 수 있는지에 대한 접근의 용이성과, 원래의 클래스 파일 크기에 대비하여 ROM 이미지 크기는 얼마나 축소되었는지에 대한 메모리 사용도를 의미한다.

5.1 접근의 효율성

ROMizer의 가장 중요한 기능 중 하나는 프리레졸루션을 통해 런타임 시 클래스의 각 항목에 대한 접근을 손쉽게 해 주는 것이다. 프리레졸루션이 없는 경우 각 항목에 대한 접근은 레졸루션을 통해 이루어지므로 <표 1>에서 밝힌 횟수 만큼의 메모리 접근이 필요하다. 반면 프리레졸루션을 이후에는 클래스를 이루는 모든 항목에 대해 균일하게 2회의 메모리 접근만으로 도달할 수 있게 된다. 따라서 상수 정보의 경우에는 동일하며, 클래스 정보에 대해서는 2배, 필드나 메소드 정보에 6배로 접근 효율성이 향상된다.

5.2 메모리 사용량

클래스 파일 구조와 (그림 4)에서 보인 ROM 이미지 내의 클래스 구조에서 가장 큰 차이는 상수 풀과 상수 풀 오프셋 테이블간의 차이에 있다. 상수 풀은 각 항목의 크기가 가변적이며 각종 링크 및 형식 확인을 위한 UTF8 문자열이 다수를 차지하고 있는데 비해 상수 풀 오프셋 테이블은 각 항목의 크기가 32비트로 고정되어있고 이들은 특정 항목의 해당 위치를 가리키는 포인터들이다. 링크와 형식 확인을 위한 UTF8 문자열은 이 테이블에 존재하지 않으며, ROMizer에 의해 프리레졸루션되어 링킹과 형식 확인은 이미 끝난 상태인 것이다.

과거 연구 결과에 따르면 데스크톱 자바 환경에서 클래스 파일의 크기 중 60퍼센트가 상수 풀에 해당되며[5], 임베디드 자바 환경에서도 절반에 근접하는 46퍼센트 정도가 상수 풀에 해당되는 것으로 조사되었다[9]. 따라서 ROM 이미지는 원래 클래스 파일 크기에 비해 절반 정도의 메모리를 사용할 것으로 예상된다.

4장에서 소개한 구조체들의 크기를 분석해보면 ROM 이미지에서 클래스 당 사용하는 메모리의 양을 수치로 나타낼 수 있다. 각 클래스는 (그림 4)의 구조를 따르므로 각 부분의 평균 크기를 예측해보면,

① 클래스 정보 : 32바이트(고정)
② 상수 풀 오프셋 테이블 : $4t$, 단 t는 상수 풀 오프셋 테이블의 길이
③ 메소드 오프셋 테이블 : $8m$, 단 m은 이 클래스가 가진 메소드의 개수
④ 상수 정보 : $(4+cl)c$, 단 cl은 상수의 평균 길이, c는 상수의 개수
⑤ 필드 정보 : $4f$, 단 f는 이 클래스가 가진 필드의 개수
⑥ 메소드 정보 : $(16+ml)m$, 단 ml은 메소드 코드 평균 길이, m 은 메소드의 개수

따라서 클래스 당 요구되는 메모리의 크기 C는 다음과 같다.

$$C = 32 + 4t + 8m + (4+cl)c + 4f + (16+ml)m$$

여기서 t는 클래스, 상수, 필드, 메소드 등 자기 클래스의 모든 항목에 대한 참조개수와 다른 클래스의 항목에 대한 참조 개수로 구성되므로 $t = 1 + c + f + m + r$(단, r은 다른 클래스 항목에 대한 참조 개수)와 같다. 이 값을 위 식에 대입하면 C는 다음과 같이 주어진다.

$$C = (36+4r) + (8+cl)c + 8f + (28+ml)m$$

simpleRTJ의 핵심 API 클래스들에 대한 분석 결과 다른 클래스의 항목에 대한 참조 개수 r = 0.5, 상수의 평균 길이 cl = 4, 상수 평균 개수 c = 1.39, 필드 평균 개수 f = 1.58, 메소드 코드 평균 길이 ml = 20.35, 메소드 평균 개수 m = 7.28 등과 같았다[9, 10]. 따라서 이 값을 위 식에 대입하면 클래스 당 요구하는 평균 메모리 크기를 알 수 있다.

C = 38 + 17 + 13 + 352 = 420바이트

이 식의 의미는 한 클래스 당 평균 420바이트의 메모리를 필요로 하며, 그 중 38바이트(9%)는 클래스 정보, 17바이트(4%)는 상수 정보, 13바이트(3%)는 필드 정보, 그리고 352바이트(84%)는 메소드 정보에 해당한다는 것이다. 바이트코드를 포함한 메소드 정보가 가장 큰 몫을 차지하고, 필드 정보가 가장 적은 몫의 메모리를 차지한다는 것을 발견할 수 있었다.

simpleRTJ의 API 클래스 파일의 평균 크기(F)가 901 바이트이고[10], ROM 이미지 내에서 한 클래스의 평균 크기(C)가 420바이트 이므로 C/F = 46.6%이다. 즉 원래 클래스 파일 크기에 비해 ROM 이미지에서는 46.6퍼센트로 줄어진 값의 메모리를 사용한다는 것이다.

실제 측정을 통한 결과 예시를 <표 2>에 나타내었다. 이 표는 simpleRTJ의 java.lang 패키지에 속한 클래스들 중 Object 등 일반 클래스 5가지와 Throwable 이하 예외/오류 클래스 5가지에 대한 측정 결과를 나타낸 것이다. 10가지 클래스 파일들의 총 크기는 13,783바이트였으며, ROMizer에 의해 변형된 후의 크기는 6,676바이트였다. 이것은 ROM 이미지 크기가 원래 파일 크기의 48.4퍼센트로 축소된 것을 말하는데, 이 측정치는 위에서 언급한 예상치 46.6퍼센트와 매우 근접한 값임을 알 수 있다.

6 결 론

ROM 기반의 임베디드 자바 시스템에서는 프로그램을 위한 다수개의 클래스 파일들이 하나의 ROM 이미지로 변형되어 탑재되어지게 되는데, 이때 ROM 이미지를 어떻게 정의하는가가 메모리의 사용량 및 클래스 내부 정보에 대한 접근 속도에 큰 영향을 미치게 된다. 본 논문에서는 자바가상기계의 원천코드가 공개되어져 있는 simpleRTJ 임베디드 자바 시스템의 ROMizer가 생성하는 ROM 이미지의 형식에 대해 분석해보았다.

ROMizer의 가장 기본적 기능 중 하나는 클래스 파일 내에 포함된 상수 풀 내의 기호명 링크 정보를 사용하여 미리 레졸루션을 수행하는 것이다. 본 논문에서는 simpleRTJ ROMizer에서 적용된 프리레졸루션에 대해 조사해보았으며, 그 결과로 만들어지는 ROM 이미지에서 클래스가 어떻게 배치되고 있는지에 대해 분석해보았다. 분석 결과 프리레졸루션으로 인해 클래스 내부 정보에 대해 최대 6배 만큼 빠른 속도로 접근이 가능해지는 것과, 메모리 사용이 평균 절반으로 줄어들 수 있음을 밝힐 수 있었다. 비록 본 연구는 simpleRTJ라고 하는 특정 시스템의 ROMizer를 대상으로 했지만 이 연구의 결과는 ROM 기반의 임베디드 자바 시스템을 위한 보다 효율적인 ROMizer의 개발에 적용되어질 수 있을 것으로 기대된다.

참 고 문 헌

[1] S. Helal, "Pervasive Java," *IEEE Pervasive Computing*, pp.82-85, Jan-Mar., 2002.

[2] D. Mulchandani, "Java for Embedded Systems," *IEEE Internet Computing*, pp.30-39, May-June, 1998.

[3] RTJ Computing, *simpleRTJ : A Small Footprint Java VM for Embedded and Consumer Devices*, http://www.rtjcom.com.

[4] T. Lindholm and F Yellin, *The Java Virtual Machine Specification*, Second Edition, Addison-Wesley, 1999.

[5] D. Antonioli and M. Pilz, "Analysis of the Java Class File Format," *Technical Report*, Dept of Computer Sci., University of Zurich, April, 1998.

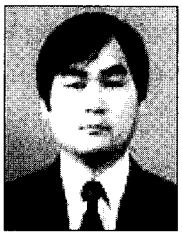
[6] 강두진, 맹혜선, 이영민, 한탁돈, 김신덕, "내장형 자바 시스템을 위한 클래스 파일의 프리레졸루션", 한국정보과학회 추계 학술발표회, pp.385-387, 1999.

[7] 양희재, "simpleRTJ 자바가상기계의 메모리 관리 기법", 한국

〈표 2〉 클래스 파일 크기와 ROM 이미지 크기의 비교 측정 결과

구 분	클래스명	파일 크기(A 바이트)	ROM 이미지 크기(B 바이트)	축소 비율(B/A%)
일 반	Boolean	1,048	496	47.3
	Integer	2,725	1,160	42.6
	Object	849	400	47.1
	String	5,705	3,012	52.8
	Thread	1,630	900	55.2
예 외/ 오 류	Throwable	625	292	46.7
	Exception	291	104	35.7
	Error	283	104	36.7
	ArithmeticException	318	104	32.7
	InternalError	309	104	33.7
전 체	10개	13,783	6,676	48.4

해양정보통신학회 춘계학술대회, pp.237-240, 2003.
[8] 양희재, "임베디드 자바가상기계를 위한 고정 크기 메모리 할당 및 해제", 대한전자공학회 하계학술대회, 제26권 제1호, pp.1335-1338, 2003.
[9] 양희재, "임베디드 자바 시스템을 위한 핵심 클래스 파일에서 상수풀 항목의 해석", 한국정보처리학회 춘계학술대회, 제10권 제1호, pp.459-462, 2003.
[10] 양희재, "simpleRTJ 클래스 파일의 형식 분석", 한국해양정보통신학회 2002 추계학술대회, 제6권 제2호, pp.373-377, 2002.



양 희 재

e-mail : hjyang@star.ks.ac.kr
1985년 부산대학교 전자공학과(공학사)
1987년 한국과학기술원 전기및전자공학과 (공학석사)
1991년 한국과학기술원 전기및전자공학과 (공학박사)
2001년~2002년 미국 펜실베니아 주립대학교 교환교수
1991년~현재 경성대학교 컴퓨터공학과 교수
관심분야 : 임베디드 시스템, 유비쿼터스 컴퓨팅, 자바가상기계