논문 2005-42CI-4-2

# 자바가상기계에서 탐침 클래스를 이용한 클래스 영역 크기의 예측

# (Estimating Size of Class Area Using Probe Classes in Java Virtual Machine)

양 희 재*

(Heejae Yang)

요 약

클래스 영역은 자바가상기계 내에서 각종 상수와 필드, 메소드 등이 위치하는 메모리공간의 한 부분이다. 임베디드 자바 시스템과 같이 제한적 메모리 자원을 갖는 시스템에서는 클래스 영역의 크기를 아는 것이 매우 중요하다. 본 논문에서는 이 영역의 크기를 예측할 수 있게 하는 일반적 수식을 유도하였다. 이 수식은 구현되는 자바가상기계에 의존적인 몇 개의 상수들을 필요로 하는데, 우리는 이들 상수들이 몇 개의 간단한 탐침 클래스에 의해 구해질 수 있음을 보였다. 본 접근 방식의 정확성을 증명하기 위한 실험 결과도 함께 나타내었다.

**Abstract**

Class area is a portion of memory where the constants, fields, and codes of the classes loaded into the Java virtual machine are kept. Knowing the size of the class area is very important especially for embedded Java system with limited memory resources. This paper induces a formula which makes it possible estimate the size of the area. The formula needs some constant values specific to target JVM implementation. We also show that these values can be found using some simple probe classes. An experimental result is included in this paper to confirm the correctness of our approach.

**Keywords**: Java virtual machine, Class file, Memory, Embedded system

## I. 서 론

잘 알려진 바와 같이 C 나 어셈블리어 등의 언어로 작성된 일반적 프로그램이 실행되기 위해서는 적어도 세 개의 메모리 영역이 필요하다[1]. 첫째는 실행될 각종 명령들이 놓이는 코드(code) 영역(또는 텍스트(text) 영역)이다. 이 영역에는 해당 컴퓨터의 기계어로 이루어진 프로그램이 놓인다. 즉치 주소 모드(immediate addressing mode)에 사용되는 숫자나 문자 등 간단한 상수도 이 영역에 위치한다.

둘째는 데이터 영역이다. 이 영역은 초기화된 데이터 영역과 초기화되지 않은 데이터 영역(bss 영역)으로 나뉘며, 각종 전역변수들과 감추어진 데이터 등이 놓여진다. 마지막 세째로 스택 영역을 들 수 있다. 이곳에는 함수 호출에 따른 리턴 주소, 파라미터 등이 저장되며 각종 지역변수들이 위치한다.

자바 프로그램도 실행되기 위해서는 몇 개의 메모리 영역을 필요로 한다. 다만 C 등으로 작성된 프로그램은 실제 CPU 상에서 해당 CPU 의 기계어로 직접 실행되는데 비해 자바 프로그램은 자바가상기계(JVM) 라는 가상의 컴퓨터 상에서 실행되며 바이트코드(bytecode)를 기계어로 사용한다는 차이가 있다.

JVM 규격에 따르면[2][3] JVM 에서 사용되는 메모리는 클래스 영역, 힙 영역, 자바 스택 영역, 그리고 네이티브 메소드 스택 영역 등 크게 네 가지의 개념적 공간으로 나뉜다 (그림 1). 여기서 클래스 영역이란 클래스에 대한 모든 정적인 정보들이 저장되는 곳이다. 즉 클래스의 이름, 상속 관계, 접근제어값, 필드와 메소드 정보 등이 모두 클래스 영역에 저장된다. 메소드를 이루는 바이트코드나 오퍼랜드로 사용되는 상수들 역시 이곳에 저장된다.

힙(heap) 영역은 객체, 즉 클래스의 인스턴스들이 놓이는 곳이다. 각 인스턴스들은 자신이 가지고 있는 필

* 평생회원, 경성대학교 컴퓨터공학과
(Department of Computer Engineering, Kyungsung University)
※ 이 논문은 2003학년도 경성대학교 학술지원연구비에 의하여 연구되었음.
접수일자: 2005년3월24일, 수정완료일: 2005년6월29일

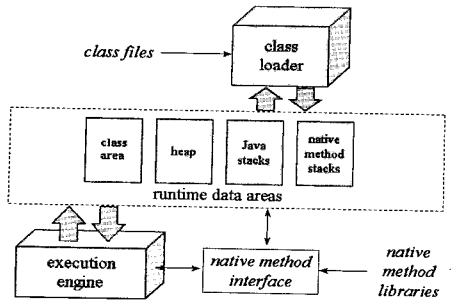



그림 1. 자바 메모리 공간
Fig. 1. Java Memory Spaces.

```
ClassFile {
    u4 magic;              // header
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count; // constant pool
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;       // class info
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;       // field info
    field_info fields[fields_count];
    u2 methods_count;      // method info
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

그림 2. 클래스 파일의 구조
Fig. 2. Structure of class file.

드는 물론 상속성의 원리에 따라 자기 조상 클래스들의 필드들을 저장할 메모리를 필요로 하는데, 힙 영역이 바로 필드 저장을 위해 사용되는 곳이다. 새로운 인스턴스가 생성될 때마다 힙 메모리가 할당되며, 그 객체가 더 이상 사용되지 않으면 쓰레기 수집기에 의해 기 할당된 힙 메모리는 회수되어진다.

자바 스택 영역은 다시 오퍼랜드 스택과 지역변수배열로 나뉘어진다. JVM 은 스택 기반의 컴퓨터이므로 모든 연산은 오퍼랜드 스택 상에 놓여진 오퍼랜드들에 대해 이루어지며, 지역변수배열은 메소드 내에서 사용되는 각종 지역변수들을 저장하는 공간이다.

마지막으로 네이티브 메소드 스택 영역이 있는데, 이곳은 네이티브 함수, 즉 C 나 어셈블리어 등 일반적 언어로 작성된 함수들이 실행될 때 필요로 하는 스택 공간을 제공한다.

유닉스 등 운영체제에서는 size 라는 명령이 제공된다. 이 명령은 프로그램의 실행 파일을 분석함으로서 그 프로그램이 실행될 때 어느 정도 메모리를 사용할 것인지를 알려주는 역할을 한다[4]. size 명령은 코드 영역과 데이터 영역의 크기를 함께 보여준다.

본 논문의 목적은 size 의 경우와 마찬가지로 JVM 에서 자바 프로그램이 실행될 때 어느 정도 크기의 메모리를 사용할 것인지 그것의 실행 파일, 곧 클래스 파일을 분석함으로서 예측하고자 하는 것에 있다.

다만 힙 영역은 객체 생성에 따라 끊임없이 증가하며, 쓰레기 수집기가 작동함에 따라 다시 정리되는 등 그 크기가 매우 가변적이므로 크기 예측의 대상이 되지 못한다. 자바 스택 역시 메소드 호출에 따라 새로 생성되며, 메소드 리턴시 사라진다는 점에서 그 크기가 동적이다.

따라서 본 논문에서는 그 크기가 항상 일정한 클래스 영역 메모리의 크기를 예측하는 것으로 범위를 정하였다. 클래스 영역의 주요 정보는 클래스 파일에 있는 내용들이므로 클래스 파일을 분석함으로서 이 영역의 크기를 예측할 수 있을 것이다. 그러나 클래스 영역 메모리에 놓이는 정보들은 클래스 파일 자체가 아니라 변형된 반사 이미지이므로[3] 클래스 파일의 분석만으로 이 크기를 예측하는 것은 매우 어려우며, 또한 구현된 JVM 플랫폼마다 예측이 달라질 수 있다.

본 논문에서는 클래스 영역의 크기를 예측하기 위한 새로운 방법을 제안하였다. 클래스 영역의 주요 정보는 클래스 정보(class), 상수 정보 (const), 필드 정보 (field), 메소드 정보 (method) 등 네 가지이다. 따라서 클래스 영역이 사용하는 메모리의 양 M 은 이 네 가지 정보들을 변수로 하는 함수 M = f(class, const, field, method) 로 나타낼 수 있다. 본 연구에서는 함수 f 가 어떤 식을 가질지 유도하고, f 내의 각 변수들이 갖는 계수(coefficient)들을 발견하기 위해 탐침 클래스를 사용하는 것을 제시하고자 한다. 이 방식을 통해 클래스 영역의 크기를 분석해 본 결과 대부분의 경우에서 메모리 사용량의 예측값과 실제 측정값이 3% 이내의 오차 범위에서 일치함을 발견할 수 있었다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 자바가상기계 명세에서 정의하는 클래스 영역 메모리에 대해 설명하며, Ⅲ장에서 클래스 영역의 크기를 나타내는 일반적 수식을 유도한다. 이 수식은 JVM 플랫폼 구현에 따라 각기 다른 값의 계수들을 가지며, 탐침 클래스를 사용하여 이 값을 밝히는 방법에 대해서 설명하였다. Ⅳ장에서는 이 수식 및 탐침 클래스를 simpleRTJ 라고 하는 특정 JVM 상에서 적용하여 본 연구의 유효성을 증명하였으며, Ⅴ장에서는 자바 메모리의 관련 연구를

소개한다. VI장에서 본 논문의 결론을 맺는다.

## II. 클래스 영역

자바에서 클래스에 대한 모든 내용은 클래스 영역 메모리에 저장되어진다. 이 영역의 구조는 클래스 파일의 구조와 매우 흡사하다. JVM 명세는 클래스 파일 내의 반사 이미지가 클래스 영역 메모리에 배치된다고 설명하고 있다.

클래스 파일은 자바 프로그램의 실행 파일에 해당되는 역할을 하며, 헤더(header), 상수 풀 (constant pool), 클래스 정보, 필드 정보, 메소드 정보 등 크게 다섯 가지 부분으로 구성된다[2].

그림 2는 JVM 명세에서 정의하고 있는 클래스 파일의 구조를 보여주고 있다. 제일 처음의 여덟 바이트가 헤더에 해당되며 매직 숫자와 버전 번호로 이루어지는 고정 영역이다.

다음에는 상수 풀 정보가 따라오는데, 이곳은 연산의 오퍼랜드로 사용되는 일반 상수들 외에도 클래스나 필드, 메소드 등의 이름과 형식 등을 나타내는 각종 상수들로 이루어진다. 후자의 상수들은 클래스 적재 시 다른 클래스들과의 링크 목적으로 사용되며, 실행 시 형식 확인 등의 목적으로도 일부 사용된다.

다음에 따라오는 access_flags 부터 interfaces[] 까지가 클래스 정보에 해당된다. 클래스의 접근제어 값, 자신 및 상위 클래스의 이름, 구현하고 있는 인터페이스의 이름 등이 클래스 정보 부분을 이룬다.

이어지는 필드 정보는 필드의 개수, 각 필드의 접근제어 값, 형식 등을 나타낸다. 마지막 부분인 메소드 정보는 메소드 개수, 각 메소드의 접근제어 값이나 형식, 반환 값의 형식 등은 물론 오퍼랜드 스택과 지역변수배열의 크기 등에 대한 정보 등을 갖는다. 자바 프로그램이 번역된 바이트코드 명령들도 바로 이 메소드 정보 속에 포함되어있다 (제일 마지막 부분의 attributes[] 에 위치한다).

클래스 파일이 메모리의 클래스 영역에 놓일 때는 다른 클래스와의 연결, 즉 레졸루션(resolution)이 완료된 상태이므로 상수 풀의 내용 중 많은 부분이 생략되어진다. 헤더 부분도 검증이 끝난 후이므로 역시 생략된다. 나머지 부분은 보다 접근하기 쉬운 형태로 변형되어 메모리에 적재된다.

클래스 파일의 구조는 JVM 명세에서 엄격하게 지정하고 있지만, 그것의 반사 이미지인 클래스 영역의 구조를 어떻게 할 것인지는 전적으로 구현하는 사람의 자유이다. 이런 자유성은 클래스 영역 메모리의 크기 예측을 어렵게 만들고 있다.

## III. 클래스 영역의 메모리 사용 분석

### 1. 클래스의 메모리 사용량

클래스 파일이 다섯 가지 부분으로 구성되는데 비해 클래스 영역에서는 헤더를 저장할 필요가 없으므로 클래스 영역 메모리는 다음과 같은 네 가지 부분으로 구성될 것임을 가정할 수 있다.

첫째는 클래스 파일 내 클래스 정보 해당 부분을 저장하는 메모리다. 이 부분은 접근제어 값, 자신 및 상위 클래스의 연결 포인터 또는 인덱스 값, 구현하는 인터페이스들에 대한 포인터, 필드 테이블과 메소드 테이블 등에 대한 포인터 등으로 이루어진다. 이 값들의 크기를 각각 산출하기는 어렵다. 예를 들어 접근제어 값의 크기는 플랫폼에 따라 16비트(short)일 수도 있으며 32비트(int)일 수도 있다. 각종 포인터나 인덱스의 크기 등도 구현된 플랫폼마다 각기 다른 값과 형식을 가질 것이며 따라서 개별 크기를 알기는 어렵다. 다만 우리가 알 수 있는 것은 이 값은 일정한 값, 즉 상수라는 것이다. 그래서 우리는 클래스 정보를 저장하는 메모리의 크기 $M_s$를 그저 어떤 상수값, 즉 $K_s$라는 값이라고 가정하자. 즉 $M_s = K_s$이다.

둘째는 상수 정보를 저장하는 메모리다. 클래스 파일의 상수 풀 내용 중에서 레졸루션을 위한 정보들은 클래스 적재 시 이미 이용된 상태이므로 이들이 클래스 영역에 있을 필요는 없다. 즉 메모리에는 바이트코드 실행 시 실제로 오퍼랜드로 사용되는 상수들만 있는 것으로 가정할 수 있다. 이들 상수들은 값과 더불어 형식(`int`, `char`, `String` 등)을 나타내는 꼬리표와 길이 등이 포함되어있으므로 $c$ 개의 상수들이 차지하는 메모리의 양 $M_c = \sum_{i=0}^{c-1} (K_c + cl_i)$ 로 나타낼 수 있다. 여기서 $K_c$는 꼬리표 등의 크기이며, $cl_i$는 $i$ 번째 상수의 길이이다. 이 식을 풀면 다음과 같이 나타내어진다. $M_c = K_c c + \Sigma cl_i$

셋째는 필드 정보를 저장하는 메모리다. 필드 정보에서 반드시 필요한 내용은 접근제어 값과 필드의 형식, 각 필드의 인덱스 값 등을 들 수 있다. 필드마다 이런 내용들이 저장되어야 하므로 메모리 사용량 $M_f$는 필드의 개수 f 에 정비례한다고 볼 수 있다. 즉 $M_f = K_f f$ 가 된다.

마지막 넷째는 메소드 정보를 저장하는 메모리이다.

메소드 정보에서 필수 내용은 접근제어 값, 형식, 스택과 지역변수배열의 크기, 바이트코드의 길이 등이다. 이 전체 정보가 요구하는 메모리 양을 $K_m$ 이라 하자. 이 값은 각 메소드의 종류에 관계없이 일정한 값을 갖는다. 다만 메소드를 이루는 바이트코드 자체의 길이는 메소드마다 각기 다르므로 $i$ 번째 메소드의 바이트코드 길이를 $ml_i$ 라고 하면 메소드 정보의 전체 메모리 사용량 $M_m$은 $M_m = \sum_{i=0}^{m-1} (K_m + ml_i) = K_m\ m + \sum ml_i$가 된다. 여기서 m 은 메소드의 개수를 의미한다.

따라서 한 클래스가 클래스 영역에서 사용하는 메모리의 사용량 $M$ 은 다음 수식으로 주어진다.

$$M = M_s + M_c + M_f + M_m$$
$$= K_s + (K_c\ c + \sum cl_i) + K_f\ f + (K_m\ m + \sum ml_i) \quad (1)$$

## 2. 탐침 클래스의 사용

JVM 내부에서 한 클래스가 사용하는 클래스 영역 메모리의 양은 식 (1) 식과 같다. 이 식에서 상수, 필드, 메소드의 개수 $c, f, m$ 상수의 길이 $cl_i$, 메소드 바이트코드의 길이 $ml_i$ 등은 클래스 파일을 분석함으로서 알 수 있다. 다만 우리가 알 수 없는 것은 계수로 사용되는 각종 상수 $K_s$, $K_c$, $K_f$, $K_m$ 등이다. 이 값들은 JVM의 구현 방식에 따라 각각의 플랫폼마다 모두 다른 값을 가지게 된다. 구현된 JVM 의 원천코드를 조사함으로서 알 수 있겠지만 코드를 모두 조사하는 것은 어려울 뿐 아니라 오류를 일으킬 확률도 크다.

본 논문에서는 이 값들을 알기 위해 탐침 클래스를 사용하는 방법을 제시하고자 한다. 즉 아주 단순한 4개의 클래스에 대해 메모리 사용량을 구한 후 식 (1) 에 대입하면 상수를 알 수 있는 것이다. 모르는 상수가 4개이므로 4개의 탐침 클래스만 사용하면 된다.

본 논문에서 사용한 탐침 클래스는 그림 3과 같다. 그림 4는 탐침 클래스들이 바이트코드로 컴파일된 내용을 보여주고 있다.

클래스 C1 은 아무런 상수나 필드, 메소드를 갖지 않는 null 클래스이다. 그러나 사실은 이와 같은 null 클래스라 할지라도 생성자 메소드를 가지므로 (그림 4 참조) $m$ = 1, $\sum ml_i$ = 5 이며, $c = f = 0$ 이다.

탐침 클래스 C2 는 한 개의 메소드를 갖는 클래스이지만, 실제로는 생성자 메소드를 포함하므로 m = 2, $\sum ml_i$ = 6 이다. 마찬가지로 c = f = 0 이다.

```
class C1 {    /* null class */
}
class C2 {    /* single method */
  void methd() {
  }
}
class C3 {    /* single field */
  C1 c;
}
class C4 {    /* single constant */
  String s = "Hello";
}
```

그림 3. 네 가지의 탐침 클래스
Fig. 3. Four probe classes.

```
class C1 extends java.lang.Object {
    C1();
}

Method C1()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

class C2 extends java.lang.Object {
    C2();
    void methd();
}

Method C2()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void methd()
   0 return

class C3 extends java.lang.Object {
    C1 c;
    C3();
}

Method C3()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

class C4 extends java.lang.Object {
    java.lang.String s;
    C4();
}

Method C4()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 aload_0
   5 ldc #2 <String "Hello">
   7 putfield #3 <Field java.lang.String s>
  10 return
```

그림 4. 탐침 클래스들의 바이트코드
Fig. 4. Bytecodes of probe classes.

탐침 클래스 C3 는 하나의 필드를 갖는 클래스이다. 즉 f = 1 이며, 생성자 메소드를 가지므로 C1 과 마찬가지로 m = 1, $\Sigma ml_i = 5$가 된다. c = 0 이다.

마지막으로 C4 는 하나의 상수를 갖는 클래스이다. 따라서 $c$ = 1 이며, $\Sigma ml_i = 5$가 된다 ("Hello" 의 길이). 이 클래스는 한 개의 필드를 가지며 ($f$ = 1), 생성자 메소드를 가지므로 $m$ = 1, $\Sigma ml_i = 11$이다 (C1 과 달리 생성자 메소드의 바이트코드 길이가 길다).

각 탐침 클래스의 메모리 사용량을 각각 $M_1$, $M_2$, $M_3$, $M_4$ 라고 하면

$$M_1 = K_s + K_m + 5 \quad (2)$$

$$M_2 = K_s + 2K_m + 6 \quad (3)$$

$$M_3 = K_s + K_f + K_m + 5 \quad (4)$$

$$M_4 = K_s + (K_c + 5) + K_f + (K_m + 11) \quad (5)$$

와 같다.

이제 실제 JVM 내에서 $M_1$, $M_2$, $M_3$, $M_4$ 의 값을 측정하면 $K_s$, $K_c$, $K_f$, $K_m$ 의 값을 알 수 있으며, 그 값을 (1)식에 대입하면 특정 JVM 내에서 일반 클래스가 사용하는 메모리의 양을 알 수 있게 된다.

## Ⅳ. 실험 및 평가

이 장에서는 실제 실험을 통해 Ⅲ장에서 설명한 탐침 클래스를 사용한 클래스 영역 메모리 사용량을 조사해 보았다. 실험에 사용한 JVM 은 RTJ computing 사의 simpleRTJ[5] 이다. 이 JVM 은 클래스 파일의 반사 이미지, 즉 클래스 영역이 ROM 에 탑재되는 모델을 취하고 있기 때문에 ROM 이미지를 분석함으로서 클래스 영역 메모리 사용량을 쉽게 측정할 수 있다는 장점이 있다.

simpleRTJ 가 제공하는 ClassLinker 라는 이름의 ROMizer를 사용하여 그림 3에서 보인 네 개의 탐침 클래스들이 차지하는 클래스 영역 메모리의 크기를 각각 측정해보았다. 측정 결과 메모리 사용량 $M_1$, $M_2$, $M_3$, $M_4$ 는 각각 68, 96, 72, 100 바이트로 조사되었다. 이 값을 (2), (3), (4) 식에 대입하면 계수 $K_s$, $K_c$, $K_f$, $K_m$ 의 값이 각각 36, 17, 4, 27 이라는 것을 알 수 있다.

이렇게 구한 계수 값들을 (1) 식에 대입하면 simpleRTJ JVM에서 클래스 영역 메모리 사용량 $M$은 다음과 같이 주어지게 된다.

표 1. simpleRTJ API 클래스들에 대한 실험
Table 1. Experiment on simpleRTJ API classes.

| 클래스 | $c$ | $\sum cl_i$ | $f$ | $m$ | $\sum ml_i$ | $M$ | 측정값 |
|---|---|---|---|---|---|---|---|
| Boolean | 2 | 9 | 3 | 8 | 155 | 462 | 496 |
| Integer | 6 | 13 | 3 | 18 | 490 | 1,139 | 1,160 |
| Object | 0 | 0 | 0 | 10 | 63 | 369 | 400 |
| String | 1 | 4 | 4 | 45 | 1,772 | 3,060 | 3,012 |
| Thread | 0 | 0 | 3 | 22 | 161 | 803 | 900 |
| Throwable | 1 | 0 | 1 | 7 | 41 | 287 | 292 |
| Exception | 0 | 0 | 0 | 2 | 11 | 101 | 104 |
| Error | 0 | 0 | 0 | 2 | 11 | 101 | 104 |
| Arithmetic-Exception | 0 | 0 | 0 | 2 | 11 | 101 | 104 |
| Internal-Error | 0 | 0 | 0 | 2 | 11 | 101 | 104 |

$$M = 36 + (17c + \sum a_i + 4f + (27m + \sum ml_i) \quad (6)$$

따라서 이제 클래스 파일의 분석을 통해 상수의 개수 $c$, 개별 상수의 길이 $cl_i$, 필드의 개수 $f$, 메소드의 개수 $m$, 개별 메소드의 바이트코드 길이 $ml_i$ 만 조사하면 simpleRTJ JVM에서 이 클래스가 클래스 영역 내에서 차지하는 메모리의 양을 예측할 수 있게 된다는 것이다. 구한 식의 정확성을 알아보기 위해 simpleRTJ 의 핵심 API 클래스들에 대해 (6) 식을 적용해 보았다. 표 1은 (6) 식을 사용한 예측 값과 실제 측정 값을 서로 비교한 것이다.

이 표에서 알 수 있듯이 거의 대부분의 경우에서 오차는 3% 이하이었으며, 다만 Boolean, Object, Thread 에서는 다소 큰 오차를 보였다. 전반적으로 (6) 식은 클래스 영역 메모리 사용량을 합당한 오차 범위 내에서 예측해 주고 있음을 발견할 수 있다.

## Ⅴ. 관련 연구

컴퓨터 시스템에서 메모리는 가장 귀한 자원 중 하나다. 특히 자원의 제약이 큰 임베디드 시스템에서는 그 중요성이 더욱 크다. 메모리는 비용, 성능, 전력소모 등 시스템 제반 분야에 많은 영향을 끼치기 때문이다. 실행 가능한 프로그램의 크기도 가용 메모리의 크기에 따라 한정되기도 한다.

이에 따라 프로그램이 요구하는 메모리의 크기를 예측하는 연구들이 많은 학자들에 의해 진행되고 있다. 특히 배열(array) 계산을 위한 프로그램의 요구 메모리 크기를 예측하는 연구가 발표되었으며[6],[7], 다중차원의 배열과 병렬성이 요구되는 멀티미디어 응용 프로그램이

필요로 하는 메모리를 예측하는 알고리즘도 연구되었다[8].

자바 메모리에 대한 연구도 활발히 이루어지고 있다. 자바 프로그램 실행에 다른 메모리 시스템 활용 형태에 대한 연구가 발표되었으며[9], 특히 클래스 영역에 놓이는 바이트코드 메소드 길이를 줄이기 위한 여러 연구가 행해졌다[10],[11],[12].

아직까지 자바 메모리, 즉 클래스 영역, 힙 영역, 자바 스택 영역의 크기 자체를 예측하는 기존 연구는 찾아볼 수 없었지만, 자바 메모리는 코드, 데이터, 스택 등 세 가지 영역으로 이루어지는 전통적 메모리 모델과 큰 차이를 보이고 있으며, 따라서 향후 자바 메모리에 대한 더욱 다양하고 깊은 연구가 필요한 실정이다[13],[14].

## VI. 결 론

하나의 자바 프로그램은 다수개의 클래스 파일로 이루어지며, 클래스 파일의 내용은 JVM 상의 클래스 영역에 저장되어진다. 클래스 파일의 내용만으로는 클래스 영역의 크기를 예측할 수가 없다.

본 논문에서는 우선 클래스 영역의 크기를 나타내는 일반적 수식을 유도하였다. 이 수식은 구현 JVM 에 따라 각기 다른 값을 갖는 네 가지 상수들을 갖는데, 이 상수 값을 알기 위해서 아주 간단한 형태의 클래스, 즉 탐침 클래스를 사용하는 방법에 대해 설명하였다. 네 가지의 각기 다른 탐침 클래스를 사용함으로서 수식에 포함된 네 가지 상수 값을 알 수 있게된다. 이렇게 하면 각 클래스가 클래스 영역 메모리를 얼마나 사용하는지를 즉시 수식을 통해 정량적으로 알 수 있게 되는 것이다. 본 연구의 결과는 특히 메모리 자원이 제한적인 임베디드 시스템의 메모리 설계에 유용하게 사용될 것으로 기대된다.

## 참 고 문 헌


[1] G. Nutt, *Operating Systems: A Modern Perspective,* 2nd ed., Addison-Wesley, 1997.

[2] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Addison-Wesley, 1997.

[3] J. Engel, *Programming for the Java Virtual Machine,* Addison-Wesley, 1999.

[4] GNU, Unix Manual Page for Size (1)

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

[6] Ying Zhao and Sharad Malik, "Exact memory size estimation for array computations without loop unrolling", *Proc of the 36th ACM/IEEE Conference on Design automation,* June 1999.

[7] Florin Balasa, Francky Catthoor, and Hugo de Man, "Exact evaluation of memory size for multi-dimensional signal processing systems", *Proc of the 1993 IEEE/ACM International Conference on Computer-Aided Design,* November 1993.

[8] Peter Grun, Florin Balasa, and Nikil Dutt, "Memory size estimation for multimedia applications", *Proc of the 6th International Workshop on Hardware/Software Codesign,* March 1998.

[9] Jin-Soo Kim and Yarsun Hsu, "Memory system behavior of Java programs: methodology and analysis", *Proc of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems,* June 2000.

[10] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller, "Java Bytecode Compression for Low-End Embedded Systems", *ACM Transaction on Programming Languages and Systems,* vol.22, no.3, May 2000, pp. 471-489

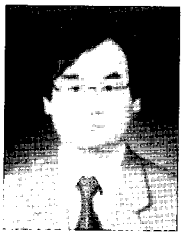
[11] J. Ramanujam, Jinpyo Hong, Mahmut Kandemir, and A. Narayan, "Reducing memory requirements of nested loops for embedded systems", *Proc of the 38th Conference on Design Automation,* June 2001.

[12] D. Rayside, E. Mamas, and E. Hons, "Compact Java Binaries for Embedded Systems", *Proc of the 9th NRC/IBM Centre for Advanced Studies Conf. (CASCON '99),* 1999, pp.1-14

[13] William Pugh, "Fixing the Java memory model", *Proc of the ACM 1999 Conference on Java Grande,* June 1999.

[14] Jeremy Manson, William Pugh, and Sarita V. Adve, "The Java memory model", *Proc of the 32nd ACM SIGPLAN-SIGACT Sysposium on Principles of Programming Languages,* January 2005.


### 저 자 소 개

양 희 재(평생회원)

1985년 부산대학교 전자공학과 (공학사)

1987년 KAIST 전기및전자공학과 (공학석사)

1991년 KAIST 전기및전자공학과 (공학박사)

1991년~현재 경성대학교 컴퓨터공학과 교수

<주관심분야 : 컴퓨터구조, 임베디드 시스템, 자바가상기계>