내장형 자바 시스템에서 클래스 파일의 분석을 통한 정적 메모리 사용량의 예측†

양희재
경성대학교 컴퓨터공학과
Email: hjyang@star.kyungsung.ac.kr

Estimation of Static Memory Usage for Embedded Java System by Class File Analysis

Heejae Yang
Dept of Computer Engineering, Kyungsung University

요 약

한 개의 자바 프로그램은 다수 개의 클래스 파일로 구성된다. 자바 프로그램이 실행되기 위해서는 클래스 파일이 메모리에 적재되어져야 하는데, 본 논문에서는 개별 클래스들이 어느 정도의 메모리를 사용할지를 정적으로 예측할 수 있게 하는 방법에 대해 알아보았다. 본 논문의 관심은 클래스 영역의 메모리, 즉 정적 메모리에 맞추어져 있으며, 힙 영역의 메모리 등 동적 메모리에 대한 예측은 향후 연구로 남겨 두었다. 클래스를 이루는 필드와 메소드 등의 값들이 메모리 사용량에 미치는 영향을 수식으로 유도하였으며, 이 수식의 유효성을 실제 실험을 통해 확인하였다.

1 서 론

한 개의 자바 프로그램은 다수 개의 자바 클래스들로 구성된다. 아주 간단한 경우를 제외하고 대부분의 의미 있는 자바 프로그램은 수백 개 또는 그 이상의 클래스들로 이루어진다. 예를 들어 썬마이크로시스템사의 HotJava 브라우저 1.1판 프로그램은 539개의 클래스로 구성되며, IDE 개발환경인 Java WorkShop 2.0판 프로그램은 1,408개의 클래스로 이루어져있다. 더 작은 규모의 프로그램인 JavaCC 컴파일러(0.7.1판)는 134개의 클래스로 이루어져있으며, 클래스 내부 정보를 알려주는 ClassViewer 등의 프로그램은 51개의 클래스로 구성되어있다 [1].

자바 프로그램의 실행을 위해 클래스들은 메모리에 적재되어져야 하며, 이 클래스들은 상당량의 메모리를 사용한다. 내장형 시스템은 메모리의 크기가 한정적이므로 너무 많은 개수의 클래스들을 메모리에 적재하는 것은 비현실적이다.

본 논문에서는 개별 클래스들이 전체 메모리 사용량에 미치는 영향에 대해 고찰해보고자 한다. 이 연구는 메모리의 크기에 제한을 받는 내장형 시스템을 위해 특히 중요하다. 일반적 데스크톱 시스템은 수백 MB 이상의 메모리를 가지며, 가상 메모리 등의 사용에 따라 클래스 적재에 따른 메모리 고갈은 거의 문제가 되지 않는다. 반면 내장형 시스템은 적은 경우 가용 메모리의 양이 수십 KB 에서 수백 KB 정도에 불과하므로 너무 많은 클래스의 수용은 어렵다.

또한 내장형 시스템은 범용 목적보다는 전용 목적을 위해 사용되는 경우가 많으며, 따라서 그 목적을 위한 프로그램이 주어진 내장형 시스템 상에서 실행

† 이 논문은 2003년도 정보통신부 지원 정보통신기초기술연구지원사업에 의해 연구되었음.

가능한 것인지를 미리 예측할 수 있게 하는 용도로 본 연구를 활용할 수 있다.

자바는 클래스 적재를 위한 정적 메모리 외에도 자바스택이나 지역변수 배열 등을 위한 동적 메모리를 필요로 한다. 후자의 동적 메모리 사용량에 대한 예측은 본 연구의 범위를 넘는 것으로 향후 연구로 남겨 두었으며, 여기서는 클래스 적재에 따른 정적 메모리 사용량에 대해서만 다루도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 자바가상기계 상의 메모리 공간에 대해 고찰해보며, 3장에서는 정적 메모리, 즉 클래스 메모리 영역에 대해 분석한다. 4장에서는 분석 결과를 실제 시스템에 대해 적용해보며, 5장에서 결론을 맺는다.

2 관련 연구

자바 프로그램은 자바가상기계(JVM) 상에서 실행된다. JVM 규격에 따르면 JVM 에서 사용되는 메모리는 다음과 같은 네 가지 개념적 공간으로 나눌 수 있다 [2].

첫째는 클래스 영역(class area)이다. 이곳에는 클래스 파일에 포함된 거의 대부분의 내용이 놓이게 되는데, 대표적인 것으로는 클래스의 이름, 상속 관계, 접근 범위, 필드 및 메소드 등이다. 바이트코드와 상수 등 중요한 정보들이 모두 여기에 포함된다.

둘째는 자바 스택 영역(Java stack area)이다. 자바에서는 메소드 호출이 일어날 때마다 스택 프레임이라는 데이터 공간이 만들어진다. 스택 프레임은 연산의 대상이 되는 오퍼랜드들이 놓여지게 되는 오퍼랜드 스택, 파라미터 전달이나 지역변수 저장을 위한 목적으로 사용되는 지역변수 배열, 그리고 프로그램 카운터 등으로 구성되어지며, 메소드 호출이 끝나면 할당된 스택 프레임은 자동적으로 사라진다.

세째는 힙 영역(heap area)으로서 각 클래스의 인스턴스들을 저장하는 목적으로 사용된다. 인스턴스들은 클래스 내의 필드들과 그 클래스의 상위 클래스들의 필드들을 저장할 수 있는 메모리 슬롯들을 필요로 한다. 인스턴스들은 new 라는 바이트코드에 의해 생성되며, 더 이상 사용되지 않는 인스턴스들에 속한 메모리는 쓰레기 수집기에 의해 반환된다.

마지막 네째는 네이티브 메소스 스택 영역(native method stack area) 으로서 C 등 언어로 작성된 네이티브 메소드가 실행될 때 사용되어지는 스택 영역이다. 이것은 일반적 컴퓨터 구조에서의 스택과 동일하다.

본 연구의 관심은 첫 번째에 언급한 클래스 영역의 크기를 예측하는 것에 있다. 이미 살펴본 바와 같이 클래스 영역은 클래스 적재 후 그 크기 및 내용이 변하지 않는 정적인 성질을 갖는다. 반면 자바스택이나 힙, 그리고 네이티브 메소드 스택 등은 메소드 호출이나 복귀, 새로운 인스턴스의 생성이나 쓰레기 수집, 네이티브 메소드의 실행 등에 따라 그 크기가 동적으로 계속 변한다.

각 영역이 차지하는 메모리의 크기는 프로그램에 따라 각각 다르다. 예를 들어 소규모 GUI 프로그램인 경우에는 클래스 영역이 가장 큰 메모리를 차지하며, 반면 대규모 서버 프로그램인 경우에는 힙 메모리가 가장 큰 크기를 차지한다 [3].

특히 힙 메모리는 개별 객체를 위한 메모리로서 프로그램 실행에 따라 동적으로 할당되고 또 해제되는 중요한 부분이다. 그 중요성 때문에 자바는 java.lang.Runtime 클래스 내에 네이티브 메소드로서 totalMemory() 와 freeMemory() 를 두고 있는데, 이들은 각각 전체 힙 메모리의 크기와 가용 힙 메모리의 크기를 반환하는 메소드이다.

이같은 기본적 도구 외에도 많은 업체들이 힙 메모리의 사용도를 분석하는 고급 프로그램을 개발하고 있다. 대표적인 것으로 썬마이크로시스템의 GC Portal 을 비롯하여 HP 의 HPjtune, 그리고 3rd party 제품군으로 JMP (Java Memory Profiler), JProfiler 등을 들 수 있다. 다만 이들 프로그램은 대부분 데스크톱 또는 서버 환경의 자바 시스템을 대상으로 하고 있다.

본 논문은 힙 메모리가 아니라 클래스 영역의 저장을 위한 메모리에 대해 연구하였으며, 특정 프로그램의 실행에 따른 동적 메모리의 사용도가 아니라 클래스 개수와 정적 메모리의 사용도 사이의 관련성에 대한 연구라는 점에서 차별성을 갖는다.

본 연구에서 밝히고자 하는 메모리는 자바 프로그램 적재를 위한 최소한의 메모리이며, 프로그램 실행시 그 크기나 내용이 변하지 않는다. 따라서 이 메모리, 즉 클래스 영역의 저장을 위한 메모리는 ROM 이나 플레시 메모리 등으로도 구현될 수 있다.

3 클래스 영역 분석

3.1 클래스 구조

클래스는 원래 클래스 파일에서 비롯된 것이므로

그 구조는 클래스 파일의 그것과 매우 흡사하다. 클래스 파일 구조는 자바가상기계 명세에서 정한 바에 따라 헤더, 상수 풀 (constant pool), 클래스 정보, 필드 정보, 메소드 정보 등 모두 다섯 개 부분으로 구성된다 [2].

헤더는 매직 번호와 버전 번호로 이루어지는 8 바이트의 고정된 영역이며, 상수 풀은 연산의 오퍼랜드로 사용되는 일반 상수 외에 클래스나 필드, 메소드 등의 이름과 형식 등을 나타내는 각종 상수들로 이루어진다. 이들 상수들은 클래스 적재 시 다른 클래스들과의 링크 목적으로 사용되며, 실행 시 형식 확인 등의 목적으로도 일부 사용된다. 클래스 정보는 이 클래스의 접근 제어값, 자신 및 상위 클래스의 이름, 구현하고 있는 인터페이스의 이름 등을 나타내며, 필드 정보와 메소드 정보는 각각 이 클래스가 갖는 필드와 메소드에 대한 정보를 가지고 있다.

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 확인이 끝난 후이므로 역시 생략된다. 나머지 부분은 보다 접근하기 쉬운 형태로 변형되어 메모리에 적재된다.

3.2 메모리 사용량

앞에서 설명한 바와 같이 클래스 파일 내용 중 일부 항목이 실제로 메모리를 사용하게 된다. 여기서는 각 항목이 사용하는 메모리의 양을 각각 분석해 보자. 당연히 이 내용은 구현 시스템에 따라 차이가 있을 수 있지만 본 연구에서는 어느 시스템에서나 필수적으로 들어갈 내용만을 지적하고, 그들이 차지하게 되는 메모리의 양을 예측해 본 것이다. 따라서 실제로는 이 값보다 다소 더 큰 크기의 메모리를 사용할 것으로 판단된다.

첫째, 클래스 정보를 이루는 요소들은 다음과 같다.

▷ 접근제어 플레그, 이 클래스의 인덱스, 수퍼 클래스의 인덱스, 필드 개수, 메소드 개수

▷ 상수 테이블, 필드 테이블, 메소드 테이블 등 각종 테이블

▷ 위 테이블을 가리키는 포인터들

각 요소 당 4바이트를 가정하면 클래스 정보의 크기는 최소 (32+각 테이블 크기) 바이트가 될 것으로 예상할 수 있다. 테이블의 각 엔트리 역시 4바이트를 차지한다고 보면 상수, 필드, 메소드의 개수를 각각 c, f, m 이라 할 때 테이블 전체가 차지하는 메모리양은 $4(c+f+m)$ 이 된다.

둘째로 상수 정보에 대해 분석해보자. 클래스 파일의 상수 풀 내용 중에서 레졸루션을 위한 정보들은 클래스 적재 시 이미 이용된 상태이므로 이들이 메모리에 있을 필요는 없다. 즉 메모리에는 바이트 코드 실행 시 실제로 오퍼랜드로 사용되는 상수들만 있다고 가정한다. 상수는 실제 값과 더불어 형식을 나타내는 꼬리표(4바이트)가 추가되므로 상수의 길이를 cl 이라 하면 상수 한 개당 $(4+cl)$ 바이트의 메모리를 사용한다.

셋째는 필드 정보이다. 필드 정보에서 필수적 내용은 접근제어 값과 필드의 형식, 그리고 개별 필드의 인덱스 값 등이다. 각각의 크기를 2바이트라고 하면 필드 한 개당 6바이트가 되며, 전체 필드 정보의 크기는 $6f$ 바이트가 된다.

넷째로 메소드 정보를 분석해보자. 메소드 정보에서 필수적 내용은 접근제어 값, 형식, 스택과 지역변수배열의 크기, 바이트코드의 길이 등이다. 각각의 크기는 4바이트라고 하면 메소드 한 개당 20바이트의 메모리를 사용하며, 여기에 실제 바이트코드가 포함되면 하나의 메소드는 $20+ml$ 바이트의 메모리를 필요로 한다. 여기서 ml 은 메소드의 바이트코드 길이다. 메소드 개수를 m, i 번째 메소드의 바이트코드 길이를 ml_i 이라고 하면 메소드 정보의 전체 크기는 $\sum_{i=0}^{m-1}(20+ml_i) = 20m+\sum_{i=0}^{m-1}ml_i$ 가 된다.

그외에도 디버깅 또는 다른 목적의 부가 정보 등이 있을 수 있지만, 내장형 시스템에서는 부가 정보가 생략되는 경우가 많으므로 본 연구에서는 고려하지 않았다.

따라서 한 클래스가 차지하는 메모리의 사용량 M 는 다음과 같은 수식으로 주어진다.

$$
\begin{aligned}
M = \ & 32+4(c+f+m) && \text{// class info} \\
& + \sum_{i=0}^{c-1}(4+cl_i) && \text{// constant info} \\
& + 6f && \text{// field info} \\
& + 20m+\sum_{i=0}^{m-1}ml_i && \text{// method info}
\end{aligned}
$$

따라서

$$M = 32 + (8c + \sum_{i=0}^{c-1}cl_i) + 10f + (24m+ \sum_{i=0}^{m-1}ml_i)$$

여기서 cl_i 는 i 번째 상수의 길이다.

만약 상수의 길이와 메소드의 바이트코드 길이에 대한 평균값을 각각 cl 과 ml 이라고 하면 평균적인 메모리 사용량 $\overline{M}$ 은 다음과 같이 주어진다.

$$\overline{M} = 32 + (8+ cl)c + 10f + (24+ ml)m$$

4 실험 및 분석

표 1은 simpleRTJ [4] 라는 내장형 자바 시스템의 핵심 API 클래스들에 대한 실험 결과를 나타낸 것이다. 예측값과 실제값의 오차는 평균 55바이트이며, 이것은 실제값의 13.2퍼센트에 해당되는 값이다. 오차의 크기가 무시할 만한 정도는 아니지만 거의 대부분의 클래스에 대해 실제값을 잘 추적하고 있음을 알 수 있다. 이러한 사실은 그림 1에서 잘 나타나고 있다.

5 결 론

하나의 자바 프로그램은 다수개의 자바 클래스들로 구성된다. 본 논문에서 우리는 개별 클래스들이 메모리 사용량에 미치는 영향에 대해 분석해보았다. 본 논문의 관심은 자바 메모리 공간 중 특히 클래스 영역의 분석에 있다. 즉 어떤 클래스가 자바가상기계에 적재되면 그 클래스가 클래스 영역의 메모리를 얼마나 차지할 것인지를 예측해보는 것이다.

고전적인 프로그램 실행 환경에서는 코드, 데이터, bss 등의 크기를 조사해봄으로서 그 프로그램이 사용할 정적 메모리의 양을 예측해 볼 수 있었다. 자바 클래스는 코드, 데이터, bss 등과 같이 구성되는 것이 아니라 필드와 메소드 등으로 구성되므로 기존 방법과는 전혀 다른 접근 방법이 필요하다. 본 논문은 클래스 파일을 분석함으로서 개별 클래스들이 얼마만큼의 정적 메모리를 소비할 것인지를 예측하게 한 것이다.

클래스 영역의 정적 메모리 뿐 아니라 향후에는 클래스 파일을 분석함으로서 힙 메모리 등 동적 메모리의 사용량도 예측할 수 있게 할 예정이다.

표 1 simpleRTJ 주요 API 클래스들에 대한 실험

클래스명	예측값	실제값	오차	
			bytes	percent
Boolean (C1)	434	496	62	12.5
Integer (C2)	1,045	1,160	115	9.9
Object (C3)	335	400	65	16.3
String (C4)	2,936	3,012	76	2.5
Thread (C5)	751	900	149	16.6
Throwable (C6)	259	292	33	11.3
Exception (C7)	91	104	13	12.5
Error (C8)	91	104	13	12.5
ArithmeticException (C9)	91	104	13	12.5
InternalError (C10)	91	104	13	12.5

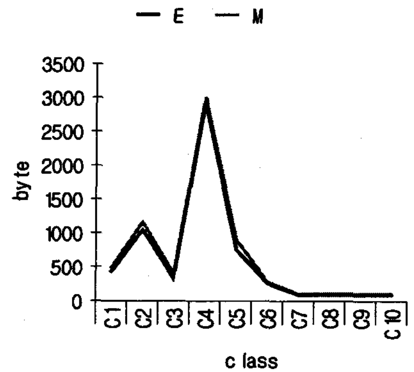

그림 1 예측값(E)과 실제값(M)의 비교

참고문헌

[1] D. Antonioli and M. Pilz, "Analysis of the Java Class File Format," TR, Dept of Computer Sci., Univ of Zurich, Apr 1998

[2] 양희재, 자바가상기계, 한국학술정보, 2001년 3월, ISBN 89-5520-342-4

[3] S. Wilson and J. Kesselman, Java Platform Performance: Strategies and Tactics, Sun Microsystems, Inc., 2000

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