자바 문장 형식이 프로그램 실행시간에 미치는 영향

양희재
경성대학교 컴퓨터공학과
e-mail:hjyang@star.ks.ac.kr

Effect of Java Statement Types on Program Execution Time

Heejae Yang
Dept of Computer Engineering, Kyungsung University

요 약

다른 고수준 언어와 마찬가지로 자바도 할당문, 연산문, 조건문, 반복문, 호출문 등의 문장 형식을 갖는다. 자바의 모든 문장 형식은 바이트코드로 변환되어 자바가상기계 상에서 실행된다. 스택 기반 구조를 갖는 자바가상기계에서 각 문장의 실행은 필연적으로 자료 이동을 요구하며, 자료 이동은 메모리 접근을 필요로 하므로 프로그램 실행 시간에 직접적 영향을 미친다. 본 논문에서는 각각의 자바 문장 형식이 어느 정도의 메모리 접근을 요구하며, 프로그램 실행 시간에는 어떤 영향을 미치는지를 분석하였다. 이 연구의 결과는 자바 프로그래머에게 프로그램 실행 시간면에서 보다 효율적인 프로그램 작성을 할 수 있도록 도와 줄 것이다.

1 서 론

모든 고수준 언어가 그렇듯이 자바 프로그램도 미리 정의된 여러 가지 문장 형식을 갖는다. 할당문 (assignment statement), 산술문(arithmetic statement), 조건문 (conditional statement), 반복문(iterative statement) 등 다양한 문장 형식이 자바 언어에서 사용되고 있다 [1].

자바에서 모든 문장은 컴파일 과정을 거쳐서 궁극적으로 바이트코드(bytecode)로 변환되고 이 바이트코드들은 자바가상기계(Java Virtual Machine)에 의해 실행된다. 본 연구의 관심은 바이트코드로 번역된 각 문장들이 JVM 상에서 실행될 때 얼마나 많은 오버헤드를 발생시키는지를 밝히는데 있다. 여기서 말하는 오버헤드는 자료 이동에 따른 시간 지연 및 에너지 소비를 의미하며, 우리는 어떤 유형의 문장 형식이 실행 시간에 가장 큰 영향을 미치는지, 즉 자료 이동이 빈번하게 일어나는지를 연구하고자 한다. 즉 어떤 유형의 문장은 다른 유형의 문장에 비해 JVM 상에서 얼마나 오버헤드 없이 (또는 자료의 이동이 최소화되면서) 실행될 수 있는지를 밝히려고 한다. 또한 동일 유형의 문장이라 하더라도 상수, 지역변수, 필드 등 어떤 자료를 사용하는지에 따라 실행시간이 크게 달라질 수 있는지도 보였다.

본 연구의 결과는 자바 프로그래머에게 프로그램의 실행 시간을 줄이기 위해 문장 선택을 어떻게 해야 할지, 프로그램의 어느 부분이 병목구간이 되며 그것을 해소할 수 있는 방법은 무엇인지, 또한 효율적 실행을 위한 자바 프로그래밍 기법은 무엇인지에 대해 정보를 제공하게 될 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 자료 이동이 일어나는 자바 메모리의 구조를 알아보는데, 필드 저장을 위한 힙, 지역변수배열, 오퍼랜드 스택 등이 핵심 메모리에 해당된다. 3장에서 자바 각 문장 형식에 따른 자료 이동 수준을 분석하며, 실행시간에 미치는 영향을 알아본다. 4장에서 연구 결과를 정리하고 결론을 맺는다.

2 자바 메모리

그림 1은 자바가상기계(JVM)의 메모리를 개념적으로 나타낸 것이다 [2] [3] [4]. 클래스 영역은 클래스 정보, 즉 메소드를 이루는 바이트코드 등이 포함되어있는 읽기 전용 영역이며, 오퍼랜드로 사용되는 각종 상수 값도 이곳에 위치한 상수 풀(constant pool)에 저장되어있다.

‡ 이 논문은 한국학술진흥재단 지역대학우수과학자 지원에 의해 연구되었음 (R05-2004-000-10967-0)

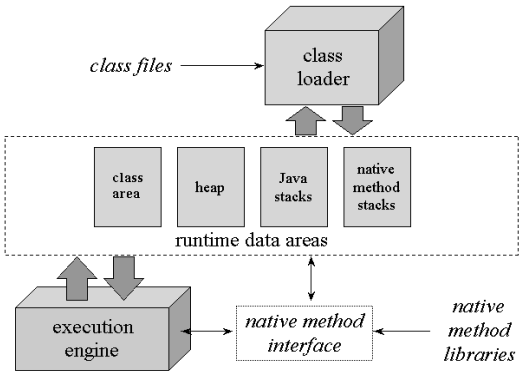

그림 1 자바 메모리

다음으로 힙(heap) 영역이 있는데, 이곳은 클래스들의 인스턴스들이 저장된다. 즉 각 인스턴스들이 가지고 있는 필드 값이 저장되며, 아울러 배열들도 이곳에 저장된다. 힙 영역은 새로운 인스턴스의 생성에 따라 할당되며, 그 인스턴스가 더 이상 사용되지 않게 되면 쓰레기 수집기에 의해 메모리가 회수되게 된다.

세 번째는 자바 스택 영역이다. 하나의 메소드가 호출될 때마다 자바 스택 영역에 스택 프레임이 새로 생성되며, 스택 프레임은 다시 오퍼랜드 스택과 지역변수배열로 나뉜다. 스택 프레임은 호출된 메소드가 종료되면 메모리에서 사라지게 된다.

마지막 네 번째는 네이티브 메소드 스택인데, 이곳은 C 등 네이티브 언어로 작성된 프로그램이 사용하는 스택 부분을 의미하며, 자바 프로그램 실행과는 큰 상관이 없으므로 본 연구에서는 제외한다.

3 문장 형식에 따른 자료 이동

모든 자바 프로그램은 자바 컴파일러에 의해 번역되며 결과는 클래스 파일에 저장된다. 클래스 파일에는 바이트코드로 변환된 자바 문장들이 저장되어있으며, JDK(Java Development Kit)가 제공하는 javap 명령을 사용하면 저장된 바이트코드를 디스어셈블(disassemble)하여 볼 수 있다.

문장 형식에 따른 자료 이동 정도를 알아보기 위해 그림 2와 같이 여러 유형의 문장으로 구성된 자바 프로그램을 작성하고 그것을 컴파일한 후 `javap` 명령으로 디스어셈블하여 분석하였다. 자바 컴파일러와 `javap` 등 자바 환경은 J2SDK 1.4.1_04를 사용하였다 [5].

```
class Test {
 int p, q;
 void f1() {
        int i, j;

        /* assignment */
        i = 0;          // loc ← const
        q = 0;          // field ← const

        i = i;          // loc ← loc
        p = i;          // field ← loc
        i = p;          // loc ← field
        p = q;          // field ← field

        /* arithmetic */
        I = i + 10;     // loc + const
        i = i + i;      // loc + loc
        i = i + p;      // loc + field
        i = p + q;      // field + field
 }
 void f2() {
        int i, j;
        boolean b;

        i = j = 0;      // dummy

        b = i > 0;      // loc & const
        b = i > 10;     // loc & const
        b = i > 50000;  // loc & const
        b = i > j;      // loc & loc
        b = i > p;      // loc & field
        b = p > q;      // field & field

        switch(i) {
        case 1:  break;
        case 2:  break;
        case 3:  break;
        default: break;
        }
 }
 /* loop */
 static final int MAX = 100;
 void f3() {
        int i;

        for (i=0; i<MAX; i++);

        i = 0;
        while (i<MAX)
                i++;
 }
 /* method invocation */
 void f4(int i, int j, int k) {
 }
 void f5() {
        int i, j, k;
        i = j = k = 0;

        f3();           // zero parameter
        f4(1, 2, 3);    // three parameters
        f4(i, j, k);    // three parameters
 }
}
```

그림 2 여러 문장 형식을 갖는 자바 프로그램

3.1 할당문

그림 2의 `f1()` 메소드에서 보인 것처럼 여러 가지 가능한 형태의 할당문에 대해 자료 이동 정도를 조사해보았다. 자료이동의 목적지는 지역변수, 필드 등이 될 수 있으며, 출발지는 지역변수(`loc`), 필드(`field`), 상수 등이 될 수 있다. 상수는 JVM 규격에 따라 8비트 또는 16비트 크기의 숫자나 문자인 경우 즉치값(`imm`)이 사용되며, 그 이상 크기의 숫자나 문자열 등은 상수풀(`cpool`)에 저장된 것을 사용한다.

표 1은 할당문에 따른 자료 이동 회수를 출발지와 목적지에 따라 정리한 것이다. 가로 항목은 출발지이며, 세로 항목은 목적지를 뜻한다.

표 1 할당문의 자료 이동 회수

	imm	cpool	loc	field
loc	1	2	2	3
field	2	3	3	4

표 1은 같은 할당문이라 할지라도 출발지와 목적지에 따라 자료 이동 회수가 최대 4배까지 차이가 날 수 있다는 것을 보여준다. 가장 적은 자료 이동은 즉치값을 지역변수에 할당하는 문장이며, 가장 많은 자료 이동은 필드값을 다른 필드에 할당하는 경우이다. 필드에 대한 접근은 지역변수에 대한 접근보다 언제나 많은 오버헤드를 일으키는 것을 발견할 수 있다. 즉 프로그램 실행시간을 줄이기 위해서는 가능한 한 필드에 대한 접근을 최소화하여야 한다는 결론을 얻을 수 있다.

3.2 산술문

그림 2의 f1() 메소드 후반부에는 두 개의 숫자를 더하는 산술문이 포함되어있다. 자바 바이트코드의 규격에 따르면 더하기 동작 뿐 아니라 빼기, 곱하기, 나누기, 논리합, 논리곱 등 모든 산술연산이 동일한 과정으로 처리되기 때문에 더하기 산술문의 오버헤드는 여타 산순문의 경우에도 동일하다.

모든 산술 연산은 오퍼랜드 스택에서 일어나므로 산술문은 필연적으로 상수나 지역변수 또는 필드의 값을 오퍼랜드 스택으로 이동하는 과정을 수반한다. 표2는 산술연산의 대상에 따라 각각 다른 수준의 자료 이동을 필요로 한다는 것을 보여주고 있다. 가로항과 세로항은 각각 산술연산의 대상이 되는 오퍼랜드를 뜻한다.

표 2 산술문의 자료 이동 회수

	imm	cpool	loc	field
imm	0	1	1	2
cpool	1	2	2	3
loc	1	2	2	3
field	2	3	3	4

즉치값끼리의 산술 연산은 가장 적은 회수의 자료 이동을 일으키며 (0회), 필드끼리의 연산은 가장 많은 회수의 자료 이동을 일으킨다 (4회). 할당문과 마찬가지로 오버헤드를 최소화하고 실행 시간을 빠르게 하기 위해서는 가능한 한 산술 연산에 필드를 개입시키는 것보다는 지역변수를 사용해야 한다는 것을 알 수 있다.

3.3 조건문

그림 2의 f2() 는 조건문, 즉 if ... else if ... else ... 와 같은 문장에 따르는 자료 이동을 발견하기 위한 것이다. 조건문에서 실제로 자료 이동을 일으키는 부분은 조건의 참과 거짓을 판단하는 부분인데, 바이트코드 수준에서 보면 이 판단은 오퍼랜드 스택에 있는 하나 또는 두 개의 값을 서로 비교하는 것이므로 필연적으로 오퍼랜드 스택으로의 자료 이동을 필요로 한다.

표3은 조건문에 따른 자료 이동 회수를 비교 대상에 따라 정리한 것이다. 가로항과 세로항은 각각 비교 대상이 되는 오퍼랜드를 뜻한다.

표 3 조건문의 자료 이동 회수

	imm	cpool	loc	field
loc	1	2	2	3
field	2	3	3	4

비교 대상에 따라 최대 4배까지의 자료 이동이 발생한다는 것을 볼 수 있다. 가장 적은 자료 이동은 즉치값을 지역변수와 비교하는 문장이며, 가장 많은 자료 이동은 필드와 다른 필드를 비교하는 경우이다. 즉 프로그램 실행시간을 최소화하기 위해서는 조건문에서 필드와 필드를 비교하는 것을 가능한 한 배제해야 하며, 지역변수와 0, 또는 16비트 이하의 크기를 갖는 숫자와 비교하는 것이 가장 권장된다고 하겠다.

f2() 의 마지막 부분은 switch 문을 사용한 조건문을 보여준다. 이 문장은 비교 대상이 되는 값이 지역변수인 경우 단 1회의 자료 이동만을 필요로 한다. switch 문 내에 case 문이 몇 개가 들어있는지에 관계없이 자료 이동은 단 1회만 일어난다. N 개의 case 문이 들어있는 switch 문을 if ... else if ... else ... 문으로 구현한다면 최대 N 회의 자료 이동이 일어나는데 비해 switch 문은 N 의 값에 상관없이 단 1회의 자료 이동만 일으킨다. 만일 비교 대상이 되는 값이 필드라면 단 2회의 자료 이동만이 요구된다.

즉 조건문을 구현할 때 자료 이동 측면에서 보면 if ... else if ... else ... 보다는 switch 문을 사용하는 편이 훨씬 효율적이며, 보다 빠른 프로그램 실행이 가능해진다.

3.4 반복문

그림 2의 f3() 는 반복문에 소요되는 자료 이동 회수를 발견하기 위한 것이다. 자바의 대표적 반복문은 for 문장과 while 문장이다.

지금까지 살펴본 바에 따르면 필드를 사용하는 것은 지역변수를 사용하는 것에 비해 항상 많은 오버헤드를 야기한다. 따라서 여기서는 반복제어변수는 지역변수를 사용하는 것으로 가정했으며, 필드를 반복제어변수로 사용하는 경우는 배제하였다. 반복제어변수의 초기값과 반복 회수는 16비트 이하의 숫자로 나타낼 수 있는 값으로 한정했다.

반복문에 대한 바이트코드를 분석해 본 결과 자료 이동은 반복 회수에 따라 달라졌다. 즉 반복 회수가 N 이라면 자료 이동은 $1+N$ 번만큼 일어난다. 처음 1번은 반복제어

변수의 초기값을 설정하기 위한 이동이며, 나머지 N번은 반복제어변수를 반복 회수와 비교하기 위해 오퍼랜드 스택으로 옮기는데 필요한 자료 이동이다. for 문이나 while 문의 자료 이동 회수는 동일했다.

for 문에서 반복제어변수를 1만큼(또는 다른 상수만큼) 증가시키는 동작은 바이트코드 iinc 에 의해 이루어지는데, 이 동작은 오퍼랜드 스택 상에서 행해지는 것이 아니라 지역변수에서 직접 행해진다. 만일 iinc 가 오퍼랜드 스택에서 일어난다면 훨씬 많은 자료 이동이 일어나게 될 것이다. 바이트코드 설계자는 iinc 동작을 자료 이동 측면에서 잘 선택한 것으로 판단된다.

하지만 자료 이동 회수가 반복 회수만큼 증가하도록 한 것은 좋은 설계가 아니다. 3.3절에서 보인 switch 문장의 경우처럼 단 1회, 또는 일정 회수의 자료 이동만이 일어나도록 새로운 바이트코드의 설계가 요구된다.

3.5 메소드 호출문

마지막으로 메소드 호출에 따른 오버헤드를 살펴보자. 그림 2의 f4(), f5() 는 메소드 호출에 소요되는 자료 이동을 발견하게 위한 것이다. f5() 에서는 파라미터(parameter)가 없는 메소드를 호출할 때, 파라미터가 있는 메소드지만 파라미터로 상수를 넘겨줄 때, 그리고 파라미터가 있는 메소드이며 파라미터로 지역변수배열을 넘겨줄 때를 각각 확인하기 위한 메소드 호출이 포함되어있다.

f5()의 바이트코드를 분석해 본 결과 파라미터가 없는 메소드를 호출하거나, 또는 파라미터가 있는 메소드이지만 넘겨주는 파라미터가 상수인 경우 1회의 자료 이동만 일어나는 것을 발견할 수 있었다. 이 자료 이동은 메소드 호출을 위한 바이트코드인 invokevirtual 이 지역변수배열의 0번째 항목인 인스턴스 포인터를 스택 상에 두는 것을 필요로 하기 때문이다.

파라미터가 있는 메소드를 호출하되 넘겨주는 파라미터가 지역변수인 경우 자료 이동은 $1+M$ 번만큼 일어난다. 여기서 M은 지역변수 파라미터의 개수이다. 메소드 호출은 자바에서 빈번히 일어나는 동작 중 하나이므로 현재의 invokrvirtual 바이트코드를 파라미터 개수와 무관하게 자료 이동이 일어날 수 있도록 새롭게 고치는 방안이 요구된다.

4 정리 및 결론

본 논문에서는 자바로 작성된 프로그램에서 각 문장 형식이 프로그램 실행 시간에 미치는 영향에 대해 연구하였다. 모든 자바 문장은 컴파일러에 의해 바이트코드로 번역되고 자바가상기계상에서 실행되는데, 이 바이트코드들이 얼마나 많은 자료 이동을 일으키는지를 분석하면 각 문장의 실행 시간을 예측할 수 있는 것이다.

연구 결과 할당문이나 산술문, 조건문은 최소 1회, 최대 4회의 자료 이동을 필요로 했다. 3가지의 문장 형식이 모두 동일한 회수의 자료 이동을 필요로 한다는 것은 무척 신기한 발견이기도 하다. 어떤 조건에서 최소의 자료 이동을 일으키는지 분석하였으며, 이 분석은 자바 프로그래머로 하여금 보다 효율적인 프로그램의 작성을 가능하게 할 것이다.

조건문의 경우 if 문장을 사용하면 조건의 개수에 비례하여 자료 이동도 증가하는데, switch 문장을 사용하면 조건 개수와 관계없이 일정량의 자료 이동이 일어난다는 점도 주목할만했다.

반복문에서는 자료 이동이 반복 회수에 1을 더한 회수만큼 일어났으며, 반복제어변수를 iinc 바이트코드를 사용하여 변경시키게 한 것은 반복문의 효율 향상에 큰 공헌을 한 설계로 확인되었다.

메소드 호출 시에는 파라미터의 개수에 따라 자료 이동 회수가 달라지는 것을 발견할 수 있었는데, switch 문장의 경우와 같이 새로운 바이트코드를 JVM 규격에 추가할 수 있다면 보다 효율적인 실행이 가능하다.

이상의 결론을 실제 자바 프로그래밍에 적용하면 시간적으로 또한 에너지 소비면에서 보다 효율적인 프로그램 개발이 이루어질 수 있다. 향후 연구는 조건문을 위한 tableswitch, lookupswitch 등의 바이트코드와 같이 반복문이나 메소드 호출을 위한 효율적인 바이트코드를 개발하고자 한다.

참고문헌

[1] C. Horstmann and G. Cornell, *Core Java 1.2*, Prentice Hall, 1999

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

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

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

[5] Sun Microsystems, *Java 2 Platform, Standard Edition (J2SE)*, http://java.sun.com/j2se