자바 프로그램에서 메모리 영역 간 자료 이동에 따른 부담 분석

(Analysis of Data Transfer Overhead Among Memory Regions in Java Program)

양 희 재 †
(Heejae Yang)

요 약 자바 프로그램이 실행되면서 자료들은 상수에서 변수로, 변수에서 변수로 등 다양한 경로로 이동한다. 자료들은 메모리에 위치하며 자료의 이동은 메모리에 대한 접근을 필요로 한다. 메모리 접근은 시간지연과 에너지 소비를 야기하므로 여러 경로의 자료 이동이 어떤 부담을 갖는지를 아는 것은 효율적 프로그램 작성은 물론 고성능 자바가상기계의 구현에도 필수적이라 할 수 있다. 본 논문에서는 자바 메모리를 상수, 지역변수, 필드 등 세 가지 영역으로 나누고 각 영역 간의 자료 이동에 대한 부담을 조사하였다. 분석 결과 지역변수에서 지역변수로의 자료 이동이 가장 부담이 작고 필드에서 필드로의 이동이 가장 부담이 큰 것으로 조사 되었으며 부담 차이는 최대 2배에 이르는 것을 발견하였다. JIT 등 최적화 기술은 자료 이동 부담을 현저히 감소시켰으며 HotSpot JVM의 경우 최소 14배에서 최대 27배까지 부담 저하 효과를 나타내었다.

키워드 : 자바, 자바가상기계, 메모리, 자료 이동, 저전력시스템

Abstract Data transfers occur during the execution time of a Java program, from constant to variable, from variable to other variable and so on. Data are located in memory and hence data transfer requires access to memory. As memory access generates both time delay and energy consumption it is absolutely necessary to know the data transfer overheads incurred among different paths not only to write an efficient program but also to build a high-performance Java virtual machine. In this paper we classify Java memory into three different regions, constant, local variable, and field, and then investigate data transfer overheads among these regions. The result says that the transfer between local variables incur the least overhead usually, while the transfer between fields incur the most. The difference of overheads reaches up to a double. Optimization techniques like JIT reduces the data transfer overhead dramatically. It is observed that the overhead is reduced from 14 to 27 times for the case of HotSpot JVM.

Key words : Java, Java Virtual Machine, Memory, Data Transfer, Low-Power System

· 이 논문은 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(R05-2004-000-10967-0)

† 종신회원 : 경성대학교 컴퓨터공학과 교수
hjyang@ks.ac.kr

논문접수 : 2007년 8월 8일
심사완료 : 2008년 3월 27일

Copyright@2008 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 소프트웨어 및 응용 제35권 제5호(2008.5)

1. 서 론

모든 컴퓨터 프로그램이 그렇듯이 자바 프로그램이 실행되면 빈번한 자료 이동이 발생된다. 즉 상수에서 변수로, 변수에서 변수로 자료들이 위치를 바꾸는 이동이 일어난다. 상수와 변수는 실행 메모리 상에 존재하므로 자료의 이동은 메모리에 대한 접근을 야기한다. 메모리에 대한 접근은 시간 지연은 물론 에너지 소비에도 영향을 미치므로 효율적 자바 프로그램의 작성 또는 고성능 자바가상기계의 개발을 위해서는 자료 이동에 대한 연구가 필수적이다[1-4].

본 논문에서는 자바 프로그램에서 일어나는 자료 이

동의 종류 및 자료 이동에 따른 부담을 조사했다. 여기서 부담이란 자료 이동에 소요되는 기계어 명령의 개수, 소요시간 등을 의미한다.

자바 프로그램에서 사용되는 변수의 종류는 메모리 상의 위치 및 사용 용도에 따라 다음과 같이 두 가지로 나눌 수 있다.

- 필드(field): 객체의 속성을 나타내는 부분으로 객체 내의 모든 메소드들이 공통적으로 사용하는 전역변수 역할을 한다. 객체가 소멸되지 않는 한 계속 메모리에 존재한다. 객체 소멸 이후 쓰레기 수집기에 의해 자동 삭제된다.
- 지역변수(local variable): 한 메소드 내에서만 사용되는 임시 변수들이다. 메소드가 호출되면 스택 프레임 상에 생성되었다가 메소드 실행이 끝나면 사라진다.

그림 1은 필드와 지역변수를 사용하는 전형적 자바 프로그램의 예를 나타낸 것이다.

이 프로그램에서 변수 balance는 필드이며 deposit(), withdraw(), isRich() 등 객체 내의 메소드들이 공통적으로 사용하는 전역변수의 역할을 한다. 반면 isRich() 메소드의 temp 는 지역변수이며 isRich() 메소드 내에서만 사용된다. 파라미터들도 지역변수다. deposit() 이나 withdraw() 메소드의 파라미터 amount 는 해당 메소드 내에서만 사용된다.

또한 자바 프로그램에서는 상수도 사용된다. Bank 클래스에서 isRich() 메소드 내의 50000 은 상수로서 자바가상기계의 상수 풀(constant pool) 영역에 저장된다. 8비트 또는 16비트로 나타낼 수 있는 작은 정수들은 상수 풀이 아니라 바이트코드 내에 포함되며, 16비트로 나타낼 수 없는 큰 정수나 실수, 문자열 등만 상수 풀에 저장된다[5].

정리하면 자바 프로그램에서는 각종 자료들이 지역변수나 필드 등 변수 형태로 존재할 수도 있으며 상수 형태로 존재할 수도 있다. 자바 프로그램 실행에 따라 자료들은 한 지역변수에서 다른 지역변수로, 한 필드에서 다른 필드로, 지역변수에서 필드로, 필드에서 지역변수로, 상수에서 지역변수로, 상수에서 필드로 등 여러 경로로 이동할 수 있는데, 본 논문에서는 이런 이동에 따른 부담을 비교 분석하고자 한다.

```
class Bank {
 int balance;   // balance is a field
 void deposit(int amount) {
       balance = balance + amount;
 }
 void withdraw(int amount) {
       balance = balance - amount;
 }
 boolean isRich() {
       // temp is a local variable
       int temp = balance - 50000;
       if (temp >= 0)
              return true;
       else
              return false;
 }
}
```

그림 1 자바 프로그램의 변수 및 상수

본 논문이 밝히고자 하는 주요 논제는 다음과 같다.

- 경로별 자료 이동에 따른 부담은 동일한가 그렇지 않은가?
- 경로별 부담이 다르다면 그 이유는 무엇인가?
- 가장 큰 부담이 따르는 경로는 무엇인가? 또한 가장 적은 부담으로 자료를 이동 할 수 있는 경로는 무엇인가?
- 전통적 인터프리터 방식 외에 JIT(Just-In-Time Compiler)[6] 등 최신 자바가상기계 기술은 경로 부담에 어떤 영향을 미치는가?
- 효율적 자바 프로그램 작성을 위해 프로그래머는 자료 이동과 관련하여 어떤 점을 유의해야 하는가?

이상의 문제를 해결하기 위해 본 논문의 2장에서는 자바 프로그램에서의 자료 이동에 대해 구체적으로 알아보고 이것이 자바가상기계 상에서 어떻게 실현되어지는지를 분석한다. 분석 결과의 확인을 위해 3장에서는 상용 자바가상기계를 사용한 실험을 하며, 인터프리터 방식 및 최적화 방식 등에서의 차이점을 조사한다. 4장에서는 본 논문의 관련 연구 및 중요성에 대해 소개하며, 마지막 5장에서 본 논문의 결론을 맺는다.

2. 자바 프로그램에서 자료 이동

서론에서 설명한 바와 같이 자바 프로그램에서 자료는 상수 형태로, 또는 지역변수나 필드에 위치할 수 있으며, 프로그램 실행에 따라 그 위치를 이동한다. 가능한 자료 이동 경로를 고찰해보면 그림 2와 같이 나타낼 수 있다.

그림 2에서 알 수 있듯이 자바 프로그램에서는 모두 여섯 가지의 자료 이동 경로가 존재한다. 상수는 값이

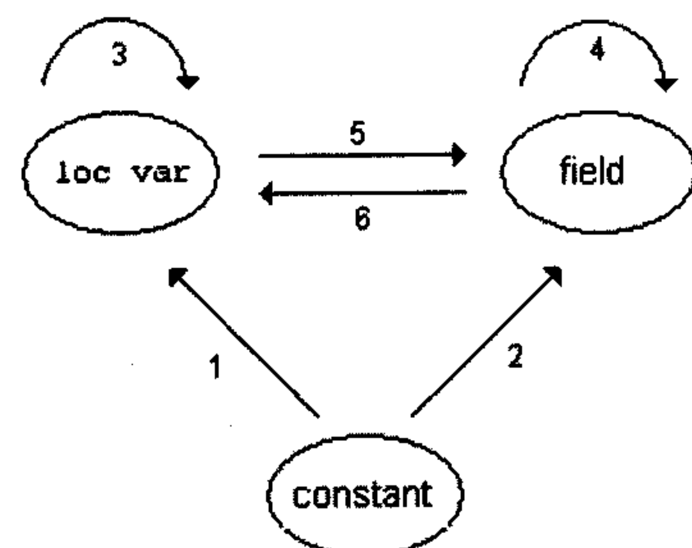

그림 2 자바 프로그램에서 자료 이동 경로

표 1 이동 경로별 바이트코드 명령

자바 문장	v1 = 40000	f1 = 50000	v2 = v1	f2 = f1	f1 = v1	v1 = f1
바이트코드	ldc #2 istore_1	aload_0 ldc #3 putfield #4	iload_1 istore_2	aload_0 aload_0 getfield #4 putfield #5	aload_0 iload_1 putfield #4	aload_0 getfield #4 istore_1
경로	1	2	3	4	5	6
	c → v	c → f	v → v	f → f	v → f	f → v

표 2 이동을 위해 사용된 바이트코드 및 의미

명령	출발지	목적지	비고
ldc	상수 풀	오퍼랜드스택	
aload	지역변수배열	오퍼랜드스택	참조자(reference) 이동
iload	지역변수배열	오퍼랜드스택	정수(integer) 이동
istore	오퍼랜드스택	지역변수배열	
putfield	오퍼랜드스택	힙	
getfield	힙	오퍼랜드스택	

일정하므로 당연히 상수로 향하는 이동 경로는 존재하지 않는다.

본 논문의 우선 관심은 그림 2에서 보인 여섯 가지 이동 경로가 모두 동일한 부담을 가지는지, 또는 그렇지 않다면 어느 경로의 자료 이동이 가장 큰 부담을 가지는지 등을 알아보는 것이다. 이것을 알아보기 위해 각 경로의 자료 이동이 바이트코드로 어떻게 번역되는지를 조사하였다. 그림 3은 지역변수(v), 필드(f), 상수(c) 사이에서 가능한 모든 자료 이동을 포함하는 자바 프로그램의 예다. 이 프로그램은 2개의 지역변수와 2개의 필드를 가진다.

그림 3의 프로그램을 컴파일 하여 각 이동 경로별로 바이트코드 명령을 조사한 결과를 표 1에 나타내었다. 표 2는 여기서 사용된 바이트코드의 종류 및 의미를 정리한 것이다.

표 2에서 언급된 상수 풀(constant pool), 오퍼랜드 스택(operand stack), 지역변수배열(local variable array), 힙(heap) 등은 자바가상기계(JVM) 내부에서 사용되는 주요 자료 구조들이다[5,7]. 자바 프로그램 상의 상수는

```
class Move {
 int f1, f2;            // fields
 void f() {
        int v1, v2;     // local vars

        v1 = 40000;     // c → v
        f1 = 50000;     // c → f
        v2 = v1;        // v → v
        f2 = f1;        // f → f
        f1 = v1;        // v → f
        v1 = f1;        // f → v
 }
}
```

그림 3 모든 자료 이동을 포함하는 자바 프로그램

상수 풀에, 지역변수는 지역변수배열에, 필드는 힙에 각각 저장된다. 오퍼랜드 스택은 연산의 대상이 되는 오퍼랜드들이 임시로 저장되는 공간으로 자바 프로그램에서 직접 접근할 수는 없다.

표 2의 바이트코드는 모두 자료 이동의 출발지와 목적지를 가지고 있다. 이 출발지와 목적지 및 바이트코드별 자료 이동 경로를 그림으로 나타내면 그림 4와 같다. 구분을 위해 그림 2의 프로그램 상 자료 이동 경로는 숫자 1~6을, 그림 4의 JVM 상 자료 이동 경로는 문자 a~e 를 각각 사용하였다.

그림 2의 프로그램 상 자료 이동은 그림 4의 JVM 상 자료 이동을 수반한다. 예를 들어 그림 2의 상수 → 지역변수(`v1 = 40000`) 자료 이동은 표 1의 바이트코드 명령에 따라 그림 4의 상수 풀 → 오퍼랜드 스택(`ldc`) 및 오퍼랜드 스택 → 지역변수배열(`istore`)의 자료 이동을 필요로 한다. 즉 그림 2의 경로 1은 그림 4의 경로 a와 c를 사용하며 자료 이동 회수는 2회다.

또한 그림 2의 상수 → 필드(`f1 = 50000`) 자료 이동은 표 1의 바이트코드 명령에 따라 그림 4의 상수 풀 → 오퍼랜드 스택(`ldc`), 지역변수배열 → 오퍼랜드 스택(`aload`), 그리고 오퍼랜드 스택 → 힙(`putfield`)의 자료 이동을 필요로 한다. 즉 그림 2의 경로 2는 그림 4

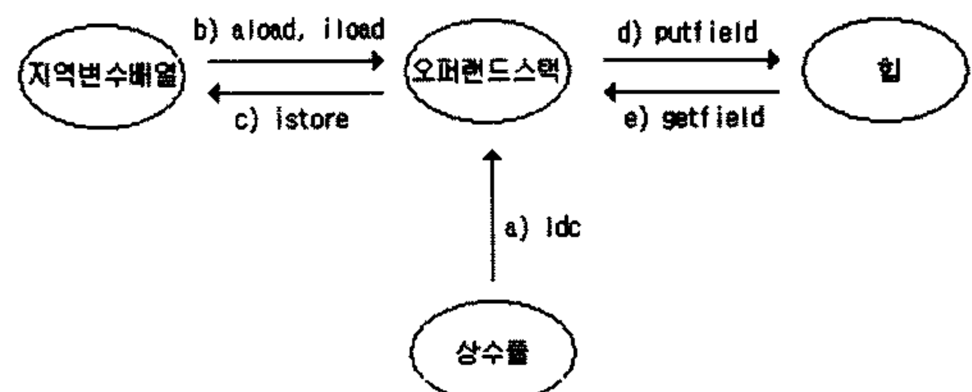

그림 4 바이트코드별 JVM 상 자료 이동 경로

의 경로 a, b, d를 사용하며 자료 이동 회수는 3회다. 여기서 경로 b 가 사용된 이유는 지역변수배열의 0번째 항목에 들어있는 객체 참조자를 오퍼랜드 스택으로 가져 오기 위함이며 JVM 규격에서 `putfield` 명령이 이 참조자를 필요로 하기 때문이다[5].

동일한 방법으로 표 1의 모든 경로에 대해 분석한 후 그 결과를 표 3에 나타내었다. 즉 표 3은 그림 2의 프로그램 상 자료 이동 경로가 어떻게 그림 4의 JVM 상 자료 이동을 수반하는지를 나타낸 것이다.

표 3에서 알 수 있듯이 자바 프로그램 상의 여섯 가지 자료 이동 경로는 JVM 상에서 각기 다른 자료 이동 회수를 필요로 한다. 예를 들어 경로 1(상수 → 지역변수)과 경로 3(지역변수 → 지역변수)은 JVM 상에서 각각 2회의 자료 이동을 필요로 하지만 경로 4(필드 → 필드)는 JVM 상에서 4회의 자료 이동을 필요로 한다.

만일 JVM 상의 자료 이동 경로 a, b, c, d, e가 모두 동일한 부담을 갖는다면 경로 1과 3의 자료 이동은 서로 동일한 부담을 가지며, 경로 4의 자료 이동은 경로 1에 비해 2배의 부담을 가진다고 볼 수 있다. 따라서 자바 프로그램에서 상수 → 지역변수 또는 지역변수 → 지역변수로의 자료 이동이 가장 적은 부담을 가지며, 필드 → 필드의 자료 이동이 가장 큰 부담을 갖는다는 것을 알 수 있다. 물론 이것은 JVM 상의 자료 이동 경로 a, b, c, d, e가 모두 동일한 부담을 갖는다는 조건에서만 성립하며, 이 조건은 사용하는 JVM 종류에 따라 맞을 수도 또는 그렇지 않을 수도 있다. 다음 장에서 여러 종류의 JVM 에 대해 실험을 통해 이 분석을 확인하고자 한다.

3. 자료이동 실험 및 분석

2장에서 우리는 자바 프로그램의 자료 이동 경로를 여섯 가지로 구분할 수 있음을 보았으며, 표 3의 마지막 줄에서 각 경로의 자료 이동에 소요되는 시간을 T_1 부터 T_6까지 각각 표기했다. 또한 JVM 상의 자료 이동 경로 a, b, c, d, e가 모두 동일한 소요시간을 갖는다면 T_1과 T_3가 가장 작은 값이며, T_4가 이 값의 두 배를 차지해 가장 큰 값을 갖는다는 것도 알아보았다.

여기서는 실제 실험을 통해 2장의 분석이 맞는지를 알아보고자 한다. 실험은 세 가지의 서로 다른 JVM, 즉 simpleRTJ(1.4.2판), HotSpot(1.5.0_06판), WebSphere Everyplace Micro Environment(5.7.2판) 등에서 각각 수행하였다.

3.1 이동 시간 측정

자료 이동에 소요되는 시간 측정을 위해 `java.lang.System` 클래스가 제공하는 `currentTimeMillis()` 메소드를 사용했다. 최근에는 더 높은 해상도를 지원하는 `nanoTime()`이라는 메소드도 제공되지만 실험에 사용한 모든 JVM이 이 메소드를 제공하는 것은 아니므로 `currentTimeMillis()` 메소드를 공통적으로 사용하였다. 즉 자료 이동 직전 시간과 직후 시간을 각각 측정하여 그 차이를 자료 이동 시간으로 산정했다.

1회의 자료 이동 소요시간은 너무나 짧기 때문에 동일한 자료 이동 동작을 N 회 반복하고 이후 총 소요시간을 N으로 나누는 방법을 사용했다. 다만 동일한 동작이 반복되면 대부분의 JVM은 자동적으로 JIT 등 최적화 모드에 진입하게 되는데, 이것을 막기 위해 명령어 라인 상 제공되는 옵션들을 사용했다. 예를 들어 HotSpot JVM 에서는 `-Xint` 옵션을 사용함으로써 JIT 모드가 아닌 인터프리터 모드를 계속 유지하게 할 수 있다. N의 값은 여러 번의 시행착오를 통해 측정 오차를 최소화하는 값으로 설정했다. 즉 HotSpot인 경우 인터프리터 모드에서는 N = 1천만을 사용했으며 동일 실험을 10회 반복하여 평균값을 구했다. JIT 모드에서는 N = 1억을 사용했다. JIT 모드인 경우 처음부터 JVM이 이 모드에서 실행되는 것이 아니므로 JIT 모드 진입 전까지의 측정값, 즉 초기 바이어스값은 제거하고 나머지 값만으로 평균값을 구하였다.

3.2 simpleRTJ 를 사용한 실험

처음 실험에 사용한 JVM은 RTJ Computing사의 simpleRTJ이다[8]. 이 JVM은 GPL 라이센스에 의해 모든 원천코드가 공개되어있다. 이것을 사용한 이유는 다른 아무런 최적화 기법을 쓰지 않은 순수한 인터프리터 방식의 엔진을 사용하고 있기 때문이다. 실험은 Windows XP Home Edition, SP2 운영체제를 사용하는 Pentium M 1.73GHz, 480MB RAM 상에서 시행했다.

경로별로 측정된 자료 이동 시간 T_1~T_6을 표 4에 나

표 3 자바 프로그램 상 자료 이동 경로와 JVM 상 자료 이동 경로

프로그램상 경로 (그림 2)	1	2	3	4	5	6
	c → v	c → f	v → v	f → f	v → f	f → v
JVM상 경로 (그림 4)	a+c	a+b+d	b+c	2b+d+e	2b+d	b+c+e
자료 이동 회수	2	3	2	4	3	3
자료 이동 시간	T_1	T_2	T_3	T_4	T_5	T_6

표 4 simpleRTJ 자바가상기계에서 경로별 자료 이동 시간

프로그램상 경로 (그림 2)	1	2	3	4	5	6	비고
	c → v	c → f	v → v	f → f	v → f	f → v	
자료 이동 시간	T_1	T_2	T_3	T_4	T_5	T_6	
simpleRTJ	174	212	153	233	188	182	nsec
	2.3	2.8	2.0	3.1	2.5	2.4	비율

타내었다. 단위는 nsec이다. 예상과 같이 T_1과 T_3는 비슷한 수준이며, T_4가 가장 크다. T_2, T_5, T_6이 T_1과 T_4의 중간 값이 되는 것도 2장의 결론과 다르지 않다.

표 4의 마지막 줄은 T_3를 기준으로 한 비율을 나타낸 것이다. T_1~T_6의 비율이 표 3과 같이 2:3:2:4:3:3과 같이 정확하게 나온 것은 아닌데, 이것은 앞에서 언급한 것처럼 경로 a~e가 모두 같은 부담을 갖는 것은 아니기 때문인 것으로 해석된다. T_1~T_6의 이론적 비율 및 실제 측정된 비율을 그림 5에 나타내었다.

이상의 실험 결과를 정리해보면 simpleRTJ JVM에 대해 다음과 같은 결론을 얻을 수 있다.

① 지역변수 → 지역변수로의 자료 이동이 가장 작은 부담을 갖는다.
② 필드 → 필드로의 자료 이동이 가장 큰 부담을 갖는다.
③ 상수 → 필드, 지역변수 → 필드, 필드 → 지역변수의 자료 이동은 중간 정도의 부담을 갖는다.
④ 상수의 접근은 지역 변수의 접근에 비해 부담이 크다(즉 그림 4의 경로 b, c에 비해 경로 a는 더 많은 부담을 갖는다).

위 결론의 ①, ②, ③은 2장의 분석과 정확히 일치한다. 결론 ④는 2장에서는 알려지지 않았지만 JVM 구현 과정에서 상수 풀에 대한 접근이 지역변수배열에 대한 접근보다 다소의 시간을 필요로 하는 까닭으로 해석된다.

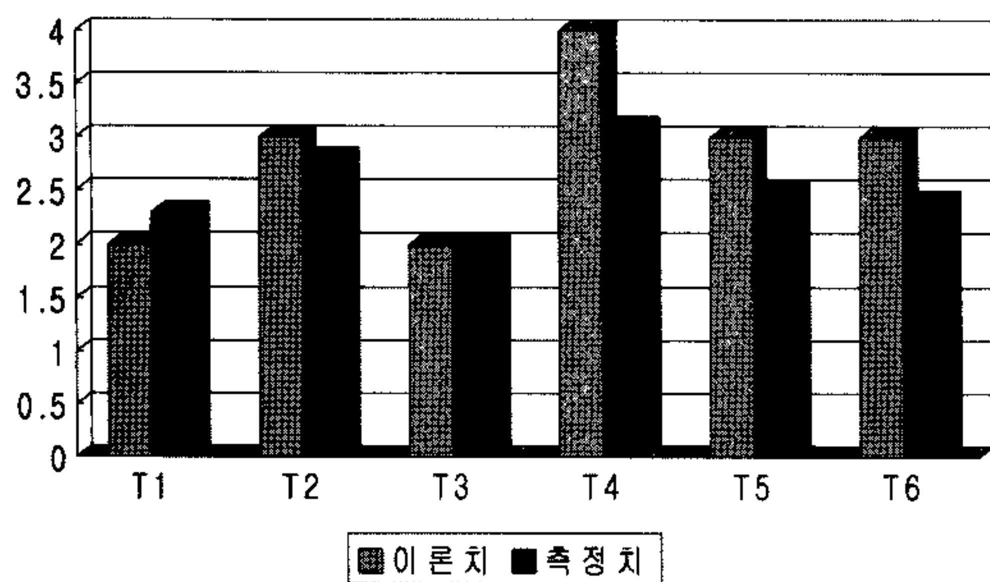

그림 5 simpleRTJ 자바가상기계에서 자료 이동 시간 비율(이론치 및 측정치)

3.3 HotSpot 및 WebSphere 을 사용한 실험

다음 실험에 사용한 JVM은 썬 마이크로시스템사의 HotSpot과 IBM 사의 WebSphere Everyplace Micro Environment이다. HotSpot은 데스크톱 환경에서 사용되는 대표적 JVM이며[9], WebSphere ME는 휴대장치인 PDA 등에서 사용되는 JVM이다[10]. 이들은 모두 널리 사용되는 상용 제품이라는 공통점을 갖고 있다.

또한 이들 JVM은 모두 인터프리터 방식 뿐 아니라 JIT 등 최적화된 방식도 함께 제공하고 있는데, 각 방식에 따른 차이점을 이해하는 것도 중요하다. 두 JVM 모두 기본적으로 최적화 모드에서 동작하며, 명령어 옵션 등을 사용하여 인터프리터 방식으로 동작시킬 수 있다. 실험은 HotSpot의 경우 Windows XP Home Edition, SP2 운영체제를 사용하는 Pentium 4 3.00GHz, 512MB RAM 상에서, WebSphere의 경우 Windows Mobile 2003(2nd Ed)을 사용하는 HP 사의 hx4700 PDA 상에서 각각 이루어졌다.

표 5에 전체 실험결과를 나타내었으며 이 표의 내용을 분석해 보면 다음과 같다(표에서 `int`는 인터프리터 모드를, `opt`는 최적화 모드를 각각 의미한다).

① 모든 경우에서 지역변수 → 지역변수로의 자료 이동이 가장 작은 부담을 갖는다.
② 필드 → 필드로의 자료 이동은 지역변수 → 지역변수로의 자료 이동에 비해 최대 2.3배의 부담을 갖는다. 이것은 2장의 해석 내용과 거의 일치하는 것이다.
③ 지역변수 → 필드, 필드 → 지역변수로의 자료 이동은 중간 정도의 부담을 갖는다.
④ 상수의 접근은 지역 변수의 접근에 비해 부담이 크며, HotSpot의 경우처럼 필드 접근보다 더 큰 부담을 갖는 경우도 있었다.

이상의 결론은 simpleRTJ에 대한 결론과 대부분 일치한다. 즉 JVM의 종류나 실행 모드와 무관하게 자료 이동에 따른 부담은 동일한 결론을 갖는 것으로 조사되었다.

따라서 효율적 자바 프로그램의 작성을 위해서 프로그래머는 다음 내용을 유념해야 할 것이다.

- JVM 종류나 모드에 관계없이 일반적으로 지역변수 → 지역변수로의 자료 이동이 가장 효율적이다.
- 필드 → 필드의 자료 이동은 대개 가장 높은 부담을 필요로 하므로 다른 우회 방법을 사용하는 편이 권장된다.
- 상수에 대한 접근은 효율적이지 않다. 따라서 동일 상수를 반복적으로 사용할 경우 상수를 어떤 지역변수로 옮겨 놓고 그 지역변수를 반복적으로 사용하는 것이 효율적이다.

표 5 HotSpot 및 WebSphere 자바가상기계에서 경로별 자료 이동 시간

프로그램상 경로 (그림 2)	1	2	3	4	5	6	비고
	c → v	c → f	v → v	f → f	v → f	f → v	
자료 이동 시간	T_1	T_2	T_3	T_4	T_5	T_6	
HotSpot int	23.5	53.1	15.6	35.9	21.9	20.3	nsec
HotSpot opt	1.4	2.0	1.1	1.9	1.4	1.4	
WebSp int	19.7	25.7	17.8	28.2	24.1	22.0	
WebSp opt	4.2	4.4	3.7	4.2	3.9	4.2	
HotSpot int	3.0	6.8	2.0	4.6	2.8	2.6	비율
HotSpot opt	2.6	3.6		3.4	2.6	2.6	
WebSp int	2.2	2.8		3.2	2.8	2.4	
WebSp opt	2.2	2.4		2.2	2.2	2.2	

• 자료 접근에 대한 부담은 일반적으로 지역변수 < 상수 < 필드의 순서다. 즉 대부분의 JVM 에서 지역변수에 대한 접근이 가장 부담이 적으며, 필드에 대한 접근 부담이 가장 크다.

3.4 사용 JVM에 따른 비교

이번에는 사용하는 JVM의 종류 및 실행 모드에 따른 차이를 비교 분석하였다. 자료의 이동 경로에 따른 접근 부담에서 가장 큰 차이를 보이는 것은 표 5에서 볼 수 있듯이 HotSpot int이다. 이 JVM의 경우 지역변수 → 지역변수의 이동 부담에 비해 필드 → 필드는 2배, 상수 → 필드는 3배 이상의 부담을 가졌다.

이와 반대로 WebSp opt는 이동 경로에 따른 접근 부담의 차이가 거의 없었다. 다른 JVM과 마찬가지로 지역변수 → 지역변수의 이동 부담이 가장 작지만, 다른 이동도 이것과 비교하여 불과 10~20% 정도의 추가 부담만 안을 뿐이다.

그러나 자료 종류에 따라 접근 부담에 차이가 없는 것이 반드시 좋은 JVM이라고 말할 수는 없다. 한 연구의 실험에 따르면 오퍼랜드 스택, 지역변수배열, 힙에 대한 메모리 접근 빈도는 각각 72%, 21%, 7% 정도였으며[11] 따라서 JVM에서는 필드 등으로의 자료 이동보다 지역변수로의 자료 이동이 월등히 많이 일어난다는 것을 알 수 있다. 또 다른 연구는 바이트코드가 실행될 때 지역변수가 접근되는 확률은 80%에 이른다고 보고하고 있다[12]. 즉 JVM 에서는 지역변수 → 지역변수의 자료 이동이 다른 이동보다 월등하게 빈번히 발생하며, 따라서 이동 경로에 따른 접근 부담의 차이를 줄이는 것 보다는 지역변수 → 지역변수의 자료 이동을 효율적으로 만든 JVM이 성능 상 오히려 좋은 것이라 볼 수 있다.

인터프리터 방식보다 JIT 등 최적화 방식을 사용한 JVM이 자료 이동 측면에서도 훨씬 효율적임을 볼 수 있다. HotSpot의 경우 인터프리터 방식보다 최적화 방식은 지역변수 → 지역변수 자료 이동에서 14배나 빨랐고, 상수 → 필드의 자료 이동은 무려 27배나 빨랐다. WebSphere의 경우도 최적화 방식은 인터프리터 방식보다 지역변수 → 지역변수 자료 이동에서 5배 빨랐고, 필드 → 필드 자료 이동은 7배가 빨랐다. 즉 효율적 자료 이동을 위해 JIT 등 최적화 기법의 도입은 매우 중요하며 효과적이라고 볼 수 있다.

4. 관련 연구 및 중요성

메모리 사용에 대한 고찰은 시간 및 에너지 측면에서 효율적 자바 프로그램 실행을 위해 매우 중요하다. 메모리는 누설전류로 인한 정적 에너지를 소비할 뿐 아니라 읽기 또는 쓰기 접근 때 마다 동적 에너지를 소비하며 시간 지연을 일으킨다.

많은 연구자들이 효율적 자바 프로그램 실행을 위한 자바가상기계의 메모리 사용에 대한 논문을 발표하였다. Chen[13,14]은 더 이상 사용되지 않는 객체의 필드를 조기에 힙 메모리에서 제거하고 그 힙 메모리를 수면 상태로 진입하게 함으로서 에너지 사용을 절약할 수 있도록 하였다. 즉 필드를 뱅크 메모리로 분산시키고 쓰레기 수집 주기를 최적화하는 알고리즘을 제안하였다.

Oi[12]는 지역변수에 초점을 맞추었는데 바이트코드 실행의 최대 80%가 지역변수에 대한 접근임을 주목하여 지역변수를 고속의 소용량 캐시 메모리에 둠으로서 성능 향상을 꾀하였다. 이 연구에 의하면 불과 16개의 지역변수 캐시만으로 최대 98%까지의 메모리 접근을 제거할 수 있었다.

Ananian[15]은 대부분의 필드가 할당된 비트 중 일부만 사용한다는 점에 착안하여 효과적으로 저장 공간의 크기를 줄이는 방법을 제안하였다. 저장 공간이 줄어들면 메모리의 정적 에너지 소비도 줄어들 뿐 아니라 자료 이동에 따른 부담도 줄어들 수 있다.

이상의 내용처럼 지금까지 발표된 논문들은 필드를 저장하는 힙 메모리, 지역변수를 저장하는 자바 스택 등 개별 메모리 공간의 효율화 방안에 대해 연구한 것이며

메모리 공간 사이의 자료 이동 부담에 대한 조사는 저자들이 알기로는 본 논문이 처음이다. 즉 지금까지 발표된 논문들은 JVM 상의 힙, 지역변수배열, 오퍼랜드 스택 등의 개별 효율성을 꾀한 것이며 이들 간의 자료 이동 효율성에 대한 연구는 현재까지 발표되지 않았다. 본 연구는 JVM 주요 메모리 공간 사이의 자료 이동 효율성에 대한 첫 시도로 볼 수 있다.

다만 현재까지 본 연구는 인터프리터 방식 및 JIT 방식 등의 채택에 따른 효율성을 조사하고 그 이유를 바이트코드 수준에서 분석 하였을 뿐이며 이동 효율성을 어떻게 높일 수 있는지에 대해서는 향후 계속적 연구를 통해 밝히고자 한다.

5. 결 론

자바 프로그래머는 프로그램의 목적에 따라 지역변수, 필드, 상수 사이에서 임의의 자료 이동을 시킬 수 있다. 본 논문은 이 세 가지 항목 사이에서 일어나는 여섯 가지 자료 이동 경로를 분석하고 각 이동 경로에 따른 부담을 조사한 것이다. 부담 조사를 위해 프로그램 상의 자료 이동이 실제 JVM 상에서 어떻게 이루어지는지를 바이트코드 수준에서 분석하였고, 경로에 따른 자료 이동 부담이 서로 다름을 밝혔다. 3가지 서로 다른 플랫폼에 대해 실험을 통해 분석 결과를 확인했다. 이 분석을 통해 우리는 자바 프로그래머가 효율적 자바 프로그램 작성을 위해 자료 이동 부담을 어떻게 고려해야 할 것임을 말했다.

또한 본 연구의 결과는 효율적 JVM 개발에 활용될 수 있을 것으로 기대된다. 자료의 이동은 메모리에 대한 접근을 필요로 하며, 메모리 접근은 시간 지연과 에너지 소비를 필요로 하므로 효율적 자료 이동은 효율적 JVM 개발에 매우 중요하다. 고전적인 인터프리터 방식에 비해 JIT 등 최적화 방식은 자료 이동 속도를 HotSpot의 경우 14배에서 27배까지 향상 시켰고 이것은 JVM 성능에 엄청난 영향을 미치고 있다. 결국 자료 이동 부담을 최소화할 수 있는 JVM 설계가 효율적 JVM 을 가능하게 하는 핵심 요소 중 하나로 볼 수 있다.

참 고 문 헌

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems," *Proc. of the 10th International Workshop on Hardware/Software Codesign (CODES'02)*, 2002.

[2] V. De La Luz, M. Kandemir, and I. Kolcu, "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems," *Proc. Design Automation Conf. (DAC'02)*, 2002, pp. 213-218.

[3] M. Mamidipaka and N. Dutt, "On-chip Stack Based Memory Organization for Low Power Embedded Architectures," *Proc. Design, Automation and Test in Europe (DATE'03)*, 2003, pp. 1082-1087.

[4] Zoe C.H. Yu, Francis C.M. Lau, and Cho-Li Wang, "Exploiting Java Objects Behavior for Memory Management and Optimizations," *Lecture Notes in Computer Science*, Springer, Volume 3302, 2004, pp. 437-452.

[5] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1999.

[6] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java just in time," *IEEE Micro*, Vol.17, No.3, May-June 1997, pp. 36-43.

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

[8] RTJ Computing Pty. Ltd., The Simple Real-Time Java, http://www.rtjcom.com/.

[9] Sun Microsystems, *The Java HotSpot Virtual Machine*, Technical White Paper, Sept 2002.

[10] IBM, *WebSphere Everyplace Micro Environment*, April 2006.

[11] 양희재, "에너지 관점에서 임베디드 자바가상기계의 메모리 접근 형태", 한국정보처리학회 논문지, 12-A권 3호, 2005. 6. pp. 223-228.

[12] H. Oi, "On the Design of the Local Variable Cache in a Hardware Translation-Based Java Virtual Machine," *Proc. ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 87-94.

[13] G. Chen, et al., "Tuning Garbage Collection in an Embedded Java Environment," *Proc. of the Int'l Symposium on High-Performance Computer Architecture*, 2002.

[14] G. Chen, et al., "Adaptive Garbage Collection for Battery-Operated Environments," *Proc. of USENIX JVM '02 Symposium*, 2002, pp. 1-12.

[15] C. Ananian and M. Rinard, "Data Size Optimizations for Java Programs," *Proc. ACM SIGPLAN Conf. on Language, Compiler, and Tool Support for Embedded Systems*, 2003, pp. 59-68.

양 희 재

1985년 부산대학교 전자공학과(공학사). 1987년 한국과학기술원 전기및전자공학과(공학석사). 1991년 한국과학기술원 전기및전자공학과(공학박사). 1991년~현재 경성대학교 컴퓨터공학과 교수. 2001년~2002년 미국 펜실베이니아 주립대학교 교환교수. 관심분야는 임베디드 시스템, 유비쿼터스 컴퓨팅, 자바가상기계