에너지 관점에서 임베디드 자바가상기계의 메모리 접근 형태

양 희 재†

요 약

임베디드 시스템에서 일반적으로 메모리가 가장 많은 에너지를 소비하는 것으로 알려지고 있다. 임베디드 자바가상기계의 경우도 마찬가지이며, 따라서 보다 에너지 효율이 높은 자바가상기계의 개발을 위해서는 자바 메모리의 에너지 사용을 최적화 하는 것이 무엇보다 중요하다. 본 논문에서는 자바 프로그램 실행 시 수많은 바이트코드들이 어떻게 논리적 메모리를 접근하는지 분석하였다. 이런 접근 형태 분석은 자바 메모리의 설계 및 구현 기술을 선택하는데 큰 통찰력을 제공해 준다. 힙, 오퍼랜드 스택, 지역변수배열 등 세 가지 논리적 데이터 공간에 대해 각각 메모리 접근을 분석하였으며, 분석 결과 오퍼랜드 스택이 가장 빈번하게, 또한 균일하게 사용되었으며 힙이 가장 드물게, 그리고 불균일하게 사용되었음을 알 수 있었다. 힙과 지역변수배열은 읽기 위주로 사용되었으며, 오퍼랜드 스택은 읽기와 쓰기 비율이 크게 다르지 않았다.

키워드 : 임베디드 시스템, 자바, 저전력 시스템

Memory Access Behavior of Embedded Java Virtual Machine in Energy Viewpoint

Heejae Yang†

ABSTRACT

Several researchers have pointed out that the energy consumption in memory takes a dominant fraction on the energy budget of a whole embedded system. This applies to the embedded Java virtual machine too, and to develop a more energy-efficient JVM it is absolutely necessary to optimize the energy usage in Java memory. In this paper we have analyzed the logical memory access pattern in JVM as it executes numerous number of bytecode instructions while running a Java program. The access pattern gives us an insight how to design and select a suitable memory technology for Java memory. We present the memory access pattern for the three logical data spaces of JVM: heap, operand stack, and local variable array. The result says that operand stack is accessed most frequently and uniformly, whereas heap used least frequently and non-uniformly among the three. Both heap and local variable array are accessed mostly in read-only fashion, but no remarkable difference is found between read and write operations for operand stack usage.

Key Words : Embedded System, Java, Low-power System

1. 서 론

휴대폰이나 PDA 와 같은 무선 이동기기에 자바 기술을 적용하는 사례가 최근 급증하고 있다. 정보 기술 분야 시장 조사 기관인 가트너 그룹(Gartner Group)은 2006년까지 전 세계 휴대폰의 80% 가 자바를 지원할 것으로 예측하고 있으며, PDA 의 경우 현재도 Palm, iPaq을 비롯한 거의 모든 PDA 들이 WabaVM, KVM, J9 VM, PERC 등의 자바가상기계를 지원하고 있다 [1] [2].

자바가 무선 이동기기를 위한 좋은 대안으로 인식되는 가장 큰 이유는 플랫폼 독립성, 즉 한번 프로그램을 작성하면 어느 환경에서나 실행시킬 수 있는 (*Write Once, Run Anywhere*) 특징 때문이다. 자바에서는 실제 프로세서 상에서 프로그램이 실행되는 것이 아니라 자바가상기계(Java Virtual Machine)라는 가상의 컴퓨터 상에서 실행되기 때문에 프로세서의 종류에 관계없는 플랫폼 독립성이 이루어지는 것이다. 그밖에도 안전성, 단순성, 동적 클래스 적재성 등이 자바를 채택하는 주요 원인이 되고 있다.

이런 무선 이동기기에서 가장 중요한 이슈 중 하나로 에너지의 소비를 들 수 있다. 이들 기기들은 모든 에너지를 배터리에서 제공받고 있으며, 빈번한 배터리 재충전 및 교체에 따른 불편을 줄이기 위해서 에너지 소비를 최소화하는 것이 무엇보다도 필요하다.

기존 연구에 따르면 휴대용 시스템에서 메모리가 가장 많은 에너지를 소비하는 것으로 알려지고 있다. 즉 시스템의 에너지 사용을 최소화하기 위해서는 무엇보다도 메모리의 에너지 사용을 최소화하여야 한다는 것이며, 이에 대한 많은 관련 연구들이 진행되고 있다[3], [4], [5], [6].

본 논문에서는 휴대폰이나 PDA 등에 탑재된 JVM 의 에너지 소비를 최소화하기 위해 JVM 실행 시의 논리적 메모

※ 이 논문은 한국학술진흥재단 지역대학우수과학자 지원에 의해 연구되었음 (R05-2004-000-10967-0)
† 정 회 원 : 경성대학교 컴퓨터공학과 교수
논문접수 : 2005년 3월 21일, 심사완료 : 2005년 6월 1일

리 접근 형태를 분석하고자 한다. 메모리가 소비하는 에너지는 메모리 접근과 관련 없이 전원이 들어오고 있는 한 누설전류에 의해 계속적으로 소모되는 정적 에너지와, 메모리를 읽거나 쓸 때 소모되는 동적 에너지로 나눌 수 있으며, 일반적으로 정적 에너지에 비해 동적 에너지가 월등히 크다 [4]. 따라서 많은 기존 논문들이 메모리의 동적 에너지를 줄이는 방법에 대해 연구하였으며, 본 논문에서도 정적 에너지가 아니라 동적 에너지의 사용에 초점을 맞추었다.

JVM 이 실행하면서 접근하는 메모리는 클래스 영역, 힙(heap) 영역, 자바 스택 영역 등으로 나누어진다[7], [8]. 자바 스택 영역은 다시 오퍼랜드 스택(operand stack)과 지역변수배열(local variable array)로 나눌 수 있다. JVM 은 클래스 영역에 놓여있는 바이트코드들을 실행하면서 끊임없이 힙 영역과 오퍼랜드 스택, 그리고 지역변수배열을 접근하게 되는데, 이 접근에 따라 메모리의 동적 에너지 소비가 일어나게 된다.

본 연구에서 우리가 밝히기 원하는 것은 JVM 의 메모리 접근 형태이다. 즉 JVM 이 실행되면서 힙과 오퍼랜드 스택, 지역변수배열 등 세 가지 논리적 메모리 영역에 대해 어느 수준으로 접근하는지를 알기 원한다. 이 세 가지 영역이 동일한 비율로 접근되는지 또는 특정 영역이 다른 영역에 비해 더 많이 접근되는지, 각 영역은 읽기 위주로 접근되는지 또는 쓰기 위주로 접근되는지 등을 분석하는 것이다.

이런 접근 형태를 알 수 있다면 보다 에너지 효율이 높은 JVM을 만드는 것이 가능해진다. 즉 빈번히 사용되는 영역과 그렇지 않은 영역의 메모리를 각기 다른 테크놀러지로 구현하고, 읽기 위주로 사용되는 메모리와 쓰기 위주로 사용되는 메모리를 각각 구분하여 필요에 따라 SPM(scratch pad memory) 이나 캐시 등 계층적 구조의 메모리로 설계함으로서 에너지 사용을 최소화시킬 수 있는 것이다 [3][9].

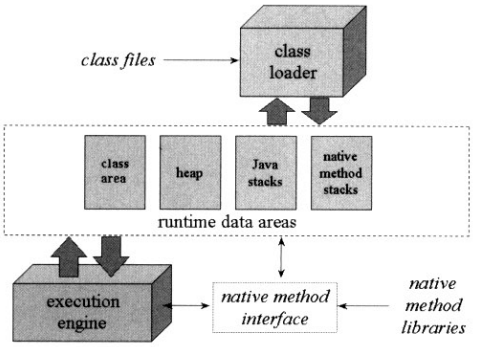
JVM 의 메모리 접근은 바이트코드(bytecode) 실행에 따라 이루어지므로 각 바이트코드가 실행되면서 어떤 형태로 메모리를 접근하는지를 분석해보면 JVM 의 메모리 접근 형태를 알 수 있다. 본 논문에서는 가장 빈번히 사용되어지는 대표적 바이트코드들에 대해 논리적 메모리 접근 형태를 분석해보고, 실험을 통해 결과를 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 JVM 의 메모리 모델에 대해 알아보며, 3장에서 대표적 바이트코드들에 대해 메모리 접근 형태를 분석한다. 4장에서는 이 분석 결과를 확인하기 위한 실험 결과를 소개하고, 5장에서는 결과 고찰과 더불어 에너지 효율적인 JVM 구현을 위한 제안에 대해 설명한다. 6장에서 본 연구를 통해 얻어진 결론을 정리한다.

2. 자바 메모리 모델

메모리는 JVM에서 매우 중요한 역할을 담당한다. 그림 1은 JVM의 대략적 구조 및 메모리 사용을 보여준다 [7] [8].

이 그림에서 알 수 있듯이 자바 메모리는 크게 네 가지의



(그림 1) 자바가상기계의 구조

논리적 공간으로 나뉜다. 먼저 클래스 영역 (class area) 이 있는데, 이곳에는 코드와 상수, 기타 클래스와 관련된 모든 정보들이 놓인다. 실행 도중에 코드나 상수 등의 정보들은 변하지 않으므로 클래스 영역은 필요에 따라 ROM 으로도 구현이 가능하다.

두 번째 부분은 힙 (heap) 영역이며 이곳에는 클래스들의 인스턴스(instances), 즉 객체들이 놓인다. 각각의 인스턴스는 클래스에서 정의한 필드들을 저장하기 위해 다수 개의 메모리 슬롯을 필요로 하며, 이 슬롯들은 힙 영역에 할당되어진다. 하나의 필드 저장을 위해 하나의 메모리 슬롯이 필요하며, 일반적으로 슬롯의 크기는 32비트이다. 상속성의 원리에 따라 자신의 부모, 조부모, 또는 그 이상 클래스들의 필드도 저장되어야 하므로 할당되는 슬롯의 수는 더욱 늘어나게 된다. 프로그램 실행에 따라 수많은 인스턴스들이 생성되며 이들을 위한 메모리 슬롯이 계속해서 할당되면 결국은 힙 영역의 메모리가 고갈되는데, 이때 쓰레기 수집기(garbage collector)가 작동하여 더 이상 사용되지 않는 인스턴스들의 슬롯을 회수하는 과정이 이루어진다.

세 번째는 자바 스택(Java stack) 영역이다. 각각의 메소드가 호출될 때마다 이 영역에 스택 프레임이라는 데이터 공간이 생성된다. 스택 프레임은 오퍼랜드 스택과 지역변수 배열, 그리고 현재 실행 중인 명령을 가리키는 포인터 등으로 구성되어져 있다. 스택 프레임은 메소드가 호출될 때 생성되며, 메소드 리턴 시 사라진다.

자바 메모리 모델의 네 번째 부분은 네이티브 메소드 스택 (native method stack) 인데, 이곳은 C 등의 언어로 작성된 네이티브 메소드 실행을 지원하기 위한 부분이다. 이 부분은 바이트코드 실행과는 무관한 곳이므로 본 논문에서는 고려하지 않았다.

3. 자바가상기계의 메모리 접근

3.1 바이트코드

바이트코드는 JVM의 기계어에 해당되는 명령어 집합이며, 총 202 가지의 명령이 제공되고 있다 [7]. 비록 이렇게 많은 명령어 종류들이 있지만, 자바 응용 프로그램에서 실

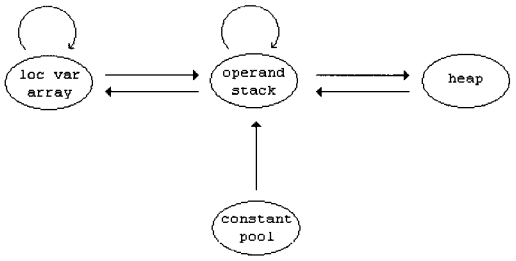
제로 사용되는 명령들은 그다지 많지 않은 것으로 알려지고 있다. 자바 프로그램은 클래스 파일에 저장되는데, 한 연구에 따르면 대표적 벤치마크 프로그램들을 담은 클래스 파일에서 발견된 바이트코드 종류는 평균 25개에 불과하였다 [10].

또 다른 연구자는 바이트코드들이 동적으로 사용되어지는 빈도에 대해 조사하였다. 이 연구에 따르면 극히 적은 수의 명령들만이 높은 빈도로 사용되어지며, 나머지 대다수 명령들은 사용도가 매우 낮은 것으로 조사되었다[11]. 지역변수 배열의 0번째 항목을 오퍼랜드 스택으로 옮기는 **aload_0** 명령이 사용된 전체 명령의 12.5%를 차지하여 가장 큰 사용빈도를 보였다. 다음은 필드값을 오퍼랜드 스택으로 옮기는 **getfield** 명령이 10.8%의 빈도를 나타내었으며, **getstatic**, **iload_1** 명령의 사용빈도는 각각 4.1%와 3.5%였다. 그외에 사용빈도가 3%를 넘는 명령은 **iload_2**(3.4%), **aload_1**(3.3%) 이었으며, 다른 명령은 모두 사용도가 3% 미만이었다. 즉 3% 이상의 사용빈도를 갖는 명령은 202개의 명령들 중 단지 6개에 지나지 않는다는 것이다.

따라서 자바 프로그램이 실행될 때 JVM 의 메모리 접근은 주로 위의 여섯 가지 명령들에 의해 이루어진다고 볼 수 있다. 즉 이들 여섯 가지 바이트코드 명령들이 메모리를 어떻게 접근하는지 분석하면 JVM 의 메모리 접근 형태를 예상할 수 있게 된다.

3.2 바이트코드 실행에 따른 메모리 접근

앞에서 살펴보았듯이 가장 빈번히 사용되는 바이트코드 명령은 **aload_0**, **getfield**, **getstatic**, **iload_1**, **iload_2**, **aload_1** 등 여섯 가지이다. 이들은 모두 자료이동 명령군에 속한다는 점에 주의할 필요가 있다. 스택 기반 구조를 갖는 JVM 에서는 거의 대부분의 동작이 스택을 경유하여 일어나므로 자료이동 명령군이 많이 사용되는 것은 어쩌면 당연한 일이다. 그림 2는 JVM 규격이 정한 모든 바이트코드 명령이 참조하는 메모리 영역 및 영역간 자료 이동을 그림으로 정리한 것이다.



(그림 2) JVM에서의 자료 이동

이 그림에서 볼 수 있듯이 자료이동의 중심은 오퍼랜드 스택이다. 가장 빈번히 사용되는 여섯 가지 바이트코드들도 오퍼랜드 스택을 경유하는데, 특히 이들 명령에서는 모두 오퍼랜드 스택이 자료이동의 목적지(destination)라는 점이 주목할만하다. 출발지(source)는 지역변수 배열 또는 힙 영역이다. 여기서 우리는 바이트코드 실행에 따른 에너지 소비를 최소화하기 위해서는 무엇보다 에너지 효율적인 오퍼랜드 스택을 구현할 필요가 있다는 것을 알 수 있다. 또한 이들 명령에서 오퍼랜드 스택은 자료 이동의 목적지이므로 읽기 동작보다는 쓰기 동작 위주로 사용되는 점에 주목해야 한다. 다음은 위 여섯 가지의 바이트코드들이 실행되면서 메모리의 어느 영역을 접근하는지에 대해 분석한 것이다.

① **aload_0, aload_1**

aload_0 와 **aload_1** 은 각각 지역변수배열의 0번째와 1번째 항목을 오퍼랜드 스택으로 옮기는 바이트코드 명령이다. 옮기는 값은 참조형(reference) 형식을 갖는다. 자바가상기계의 규격에 따르면 이 명령은 최소 2회의 메모리 접근을 필요로 함을 알 수 있다. 즉 한번은 지역변수배열을 읽는 것이며, 다른 한번은 오퍼랜드 스택의 제일 위에 값을 쓰는 것이다.

② **iload_1, iload_2**

iload_1 과 **iload_2** 는 각각 지역변수배열의 1번째와 2번째 항목을 오퍼랜드 스택으로 옮기는 바이트코드 명령이다. 옮기는 값의 형식이 정수형이라는 점 외에는 앞의 **aload_0** 등과 같으므로 소요되는 메모리 접근 회수는 마찬가지로 2회이다.

③ **getfield, getstatic**

getfield #i 는 어떤 객체의 **i**번째 필드(인스턴스 변수)를 가져와서 오퍼랜드 스택의 제일 위로 이동시키는 바이트코드 명령이다. 자바가상기계 규격에 따르면 이 명령 실행 시 객체를 가리키는 참조자는 오퍼랜드 스택의 제일 위에 있기 때문에 먼저 이 참조자를 읽어와야 한다. 따라서 이 명령은 최소 3회의 메모리 접근을 필요로 한다. 첫째는 객체 참조자를 얻기 위해 오퍼랜드 스택을 읽는 것이고, 둘째는 필드를 가져오기 위해 힙 메모리를 읽는 것이며, 마지막 셋째는 그 값을 오퍼랜드 스택에 두기 위해 오퍼랜드 스택에 쓰는 것이다.

getstatic #i 는 어떤 클래스의 **i**번째 정적 필드(클래스 변수)를 가져와서 오퍼랜드 스택의 제일 위로 이동시키는 바이트코드 명령이다. 이 명령은 **getfield** 명령과 달리 객체를 가리키는 참조자를 얻을 필요가 없으므로 2회의 메모리 접근으로 실행된다. 한번은 필드를 가져오기 위해 힙 메모리를 읽는 것이며, 다른 한번은 오퍼랜드 스택의 제일 위에 두기 위해 오퍼랜드 스택에 쓰는 것이다.

3.3 요약

<표 1>은 일반적 자바 프로그램 실행 시 가장 빈번히 사용되는 여섯 개의 바이트코드 실행에 따른 메모리 접근 정도를 정리한 것이다. 이 표에서 각각의 숫자는 메모리 접근의 횟수를 나타내며, R과 W는 각각 읽기와 쓰기 동작을 나타내고 있다. 이 표는 또한 힙 메모리와 지역변수배열은 주

〈표 1〉 주요 바이트코드의 메모리 접근 형태

바이트코드	힙	자바 스택		접근 횟수	사용빈도 (%)
		오퍼랜드 스택	지역변수배열		
aload_0		1W	1R	2	12.5
aload_1		1W	1R	2	3.3
iload_1		1W	1R	2	3.5
iload_2		1W	1R	2	3.4
getfield	1R	1R, 1W		3	10.8
getstatic	1R	1W		2	4.1

로 읽기 위주로 사용되고 있으며, 오퍼랜드 스택은 쓰기 위주로 사용되고 있음을 보여주고 있다. **aload_*n***, **iload_*n***, **getstatic** 등 명령은 각각 두 번의 메모리 접근을 필요로 하며, **getfield** 는 세 번을 필요로 한다.

4. 실험 및 분석

3장에서 얻은 결론은 힙 메모리와 지역변수배열은 주로 읽기 위주로 사용되며, 오퍼랜드 스택은 쓰기 위주로 사용된다는 것이다. 또한 메모리 영역 중 오퍼랜드 스택이 가장 빈번히 접근되며, 반면 힙 메모리는 가장 적게 사용된다는 것도 추론할 수 있었다. 이와 같이 바이트코드 동작의 분석을 통해 얻어진 결론을 확인하기 위해 실제 실험을 시행하였다.

4.1 실험 환경

실험을 위한 JVM은 RTJ Computing 사의 임베디드 자바가상기계인 simpleRTJ이며, 이것은 8비트, 16비트, 및 32비트의 다양한 프로세서와 매우 적은 메모리 환경을 지원하는 원천코드가 공개되어있는 JVM이다[12]. 또한 벤치마크 프로그램으로 simpleRTJ API 클래스 라이브러리에서 발견되는 주요 메소드를 담은 별도의 프로그램을 개발하여 사용하였다. 비록 SPECjvm98[13] 을 비롯한 많은 자바 벤치마크 프로그램들이 알려져 있지만 이들의 목적은 대개 데스크톱 환경의 JVM 들에 대한 성능 평가를 위한 것들로 본 연구에서 관심을 갖는 임베디드 JVM 환경과는 많은 차이가 있다. 예를 들어 임베디드 시스템은 일반적으로 실수형 자료를 다루지 않고 정수형만 다루므로 SPECjvm98 의 몇몇 평가 프로그램은 적절한 벤치마크가 되기 어렵다.

우리가 사용한 벤치마크 프로그램은 주요 API 클래스에서 발견되는 메소드들을 포함한 것으로 47개의 서로 다른 클래스가 사용되었으며 다음과 같은 기능을 갖는다.

- 주요 객체의 사용 (Integer, String 등 주요 클래스 생성 및 사용)
- 다중 쓰레드 프로그램
- 파일 입출력
- StringTokenizer

4.2 바이트코드 및 메모리 사용도

벤치마크 프로그램에서 실행된 13,974개의 바이트코드 중에서 가장 빈번히 사용된 상위 10개의 바이트코드 명령들을

〈표 2〉 가장 빈번히 사용된 바이트코드

순위	바이트코드	실행횟수	실행비율(%)
1	aload_0	1,844	13.5
2	getfield	1,432	10.2
3	iload_1	709	5.1
4	iload_2	607	4.3
5	if_icmplt	593	4.2
6	iconst_0	475	3.4
7	iinc	458	3.3
8	invokevirtual	408	2.9
9	aload_1	401	2.9
10	iadd	375	2.7

<표 2>에 나열하였다. 기존 연구[11]에서 조사된 결과와 표 2의 결과가 거의 일치함을 발견할 수 있다. 다만 **getstatic** 명령은 기존 연구와 달리 여기서는 많이 사용되지 않았는데, 우리가 사용한 벤치마크 프로그램에서는 클래스 메소드 사용이 상대적으로 적었기 때문인 것으로 해석된다.

또한 벤치마크 프로그램 실행에 따라 모두 30,696번의 메모리 접근이 이루어졌으며, 각 영역별 접근 회수를 표 3에서 정리하였다. 이 표에서 발견할 수 있듯이 대부분의 메모리 접근은 오퍼랜드 스택에 집중되었으며(71.4%), 지역변수배열에 대한 접근도 꽤 높은 편이었다(21.2%). 그러나 힙 영역에 대한 접근은 상대적으로 작은 편이다(7.4%). 이 결과는 3장에서 얻었던 분석 결과와 정확히 일치하고 있다. 즉 JVM 의 메모리 접근 형태는 가장 빈번히 사용되는 여섯 가지 바이트코드의 메모리 접근 형태와 거의 동일하다는 것이다.

전체적으로 볼 때 메모리에 대한 읽기와 쓰기 비율은 6:4임을 <표 3>에서 볼 수 있다. 그러나 각 영역에서 그 비율은 현저한 차이를 나타내고 있다. 즉 힙 메모리와 지역변수배열의 경우 그 비율은 8:2 정도로 나타났다. 다시 말하면 힙 영역과 지역변수배열은 주로 읽기 위주로 사용된다는 것이다. 그러나 오퍼랜드 스택의 경우는 읽기와 쓰기 비율이 6:4 로서 두 가지 동작의 차이가 그리 큰 편은 아니다. 이와 같은 결론도 3장에서 얻었던 분석 결과와 일치함을 알 수 있다.

〈표 3〉 메모리 영역별 읽기/쓰기 횟수

영역	읽기	쓰기	합계	비율(%)
힙	1,844	433	2,277	7.4
오퍼랜드 스택	12,059	9,851	21,910	71.4
지역변수배열	5,381	1,128	6,509	21.2
합계	19,284	11,412	30,696	100.0

4.3 시간 분포면에서의 메모리 접근 형태

이번에는 시간 분포면에서 메모리 접근 형태를 분석해보았다. <표 4>는 벤치마크 프로그램 시작부터 종료 때까지 매 1,000개씩의 바이트코드가 실행될 때마다 메모리 영역별 읽기/쓰기 회수를 정리한 것이다.

이 표를 보면 오퍼랜드 스택은 시간대와 관계없이 항상 활발히 사용됨을 알 수 있다. 즉 바이트코드 명령이 1,000회 실행될 때 오퍼랜드 스택에 대한 읽기 동작은 평균 861회

〈표 4〉 시간대별 메모리 읽기/쓰기 횟수

바이트코드 실행	힙		오퍼랜드 스택		지역변수배열	
	읽기	쓰기	읽기	쓰기	읽기	쓰기
1,000	71	29	946	630	407	52
2,000	170	49	899	752	431	47
3,000	111	100	708	630	455	118
4,000	86	29	860	686	312	115
5,000	86	29	855	686	316	113
6,000	86	28	856	687	314	115
7,000	66	32	873	670	358	95
8,000	63	31	815	688	375	93
9,000	62	31	811	689	374	94
10,000	104	29	857	691	392	83
11,000	238	11	896	768	420	52
12,000	234	11	909	765	411	49
13,000	236	11	901	763	412	51
14,000	231	13	873	746	404	51
합계	1,844	433	12,059	9,851	5,381	1,128
평균	132	31	861	704	384	81
표준편차	73	23	57	47	45	29

일어나며, 쓰기 동작은 평균 704회 일어난다. 표준편차는 각각 57, 47로서 평균값에 비해 매우 작은 값이다. 즉 오퍼랜드 스택에 대한 접근 형태는 매우 균일한 것을 알 수 있다.

반면 힙 메모리에 대한 접근은 시간대에 따라 꽤 많은 차이를 보였다. 바이트코드 명령이 1,000회 실행될 때 어떤 시간대에는 62회의 읽기 동작이 일어났지만, 어떤 시간대에는 그것의 3.8배에 해당되는 238회의 읽기 동작이 일어났다. 1,000회 당 읽기 동작의 평균값은 132회이지만 표준편차가 73으로서 평균값에 비해 매우 큰 값이다. 또한 쓰기 동작은 더 큰 편차를 보였는데, 어떤 시간대에는 11회의 쓰기 동작이 이루어진 반면 다른 시간대에는 그것의 9.1배에 해당되는 100회의 쓰기 동작이 일어났다. 쓰기 동작에 대한 평균은 31회였으며, 표준편차는 23으로서 평균값에 비해 매우 큰 값이다. 즉 힙 메모리에 대한 접근 형태는 매우 불균일하며, 특정 시간에 돌발적으로 증가하는 (burst) 형태를 나타내었다.

지역변수배열도 오퍼랜드 스택과 마찬가지로 비교적 균일한 접근 분포를 보였다. 바이트코드 명령이 1,000회 실행될 때 지역변수배열에 대한 읽기 동작은 평균 384회 일어나며, 쓰기 동작은 평균 81회 일어난다. 표준편차는 각각 45, 29로서 역시 평균값에 비해 작은 값을 나타내었다.

5. 분석결과와 고찰

5.1 논리적 메모리 접근 형태

우리는 202가지의 바이트코드 중 3% 이상의 사용 빈도를 갖는 명령이 불과 6개에 지나지 않는다는 사실에 착안하여 [11] 이 6가지 바이트코드의 메모리 접근 형태를 자바가상기계 규격에 따라 분석하였으며, 그 결과 오퍼랜드 스택이 가장 빈번히 사용되며 힙이 가장 드물게 사용되는 것을 추론할 수 있었다. 또한 힙과 지역변수배열은 읽기 위주로 사용된다는 것도 알 수 있었다(표 1). 실제 실험을 통해 얻은 결론 역시 처음 예상과 일치하였으며(표 3), 거의 모든 바이트코드 실행 때마다 오퍼랜드 스택에 대한 접근이 일어남도 발견할 수 있었다.

메모리에 대한 접근은 하드웨어적으로 동적 에너지의 소비를 필요로 하며, 따라서 에너지 효율적 JVM의 구현을 위해서 본 연구에서 얻은 결론은 다음과 같다.

- 오퍼랜드 스택을 특히 에너지 효율이 높은 메모리로 구현하여야 한다. JVM 실행 시 메모리 접근의 70% 이상이 오퍼랜드 스택으로 집중되어지는 것을 발견할 수 있었다.
- 반면 힙 메모리는 사용 빈도가 낮기 때문에 사용되지 않는 기간 동안 메모리를 비활성화 시켜서 사용 전력을 줄이는 전략이 필요하다. 최근의 DRAM은 활성화(active), 대기(standby), 수면(napping), 차단(power-down), 불능(disabled) 등 다양한 전력 모드를 제공하고 있으며 [4], 힙 메모리의 동작 모드를 필요에 따라 변화시키는 것이 에너지 효율을 높일 수 있다.
- 힙 메모리와 지역변수배열은 읽기 위주로 사용되고 있기 때문에 캐시 또는 SPM(scratch pad memory) 등을 적극 사용할 필요가 있다. 읽기 위주로 사용될 경우 캐시 적중 비율이 매우 높기 때문에 메인 메모리의 일부를 비활성화하는 등의 에너지 절감이 가능하다.
- 거의 모든 메모리 접근이 오퍼랜드 스택으로 집중되기 때문에 인터프리터 방식이 아닌 JIT(Just-in-time) 컴파일러 방식 등을 사용하여 실행하는 편이 에너지 절감에 효과적일 수 있다. JIT 방식은 오퍼랜드 스택의 사용을 부분적 또는 전부 제거할 수 있기 때문이다.

5.2 메모리 비활성화의 적용 가능성

4.3절에서 밝힌 바와 같이 힙 메모리에 대한 접근은 매우 불균일하며, 특정 시간대에 돌발적으로 집중되는 형태를 보이고 있다. 이런 특징은 힙이 사용되지 않는 기간 동안 해당 영역 메모리를 비활성화시켜 에너지 소비를 감소시킬 수 있는 가능성을 보여준다. 예를들어 표 4의 시간대 중 11,000회 실행 이후에는 바이트코드 1,000회 실행 당 11번, 즉 100회 당 1번 정도의 힙 메모리 쓰기 동작이 일어나는 것을 볼 수 있다.

하나의 바이트코드 실행은 그 종류에 따라 꽤 많은 시간을 필요로 한다. 예를들어 두 개의 정수형 값을 서로 더하는 iadd 바이트코드를 simpleRTJ 는 다음과 같은 함수로 구현하고 있다.

```
void iadd() {
    vm_sp -= 2;
    vm_sp[0].i += vm_sp[1].i;
    vm_sp++;
}
```

이 함수를 gcc 3.2.1 컴파일러를 사용하여 번역하면 총 26개의 ARM 프로세서 기계어 실행이 필요함을 알 수 있는데,

위 함수 실행 외에 인터프리터 실행에 따른 추가부담도 포함되어 하나의 바이트코드 실행에 소요되는 시간은 더욱 길어질 것으로 예상된다. 따라서 100회의 바이트코드 실행은 적어도 5,000~10,000 정도의 메모리 사이클을 필요로 할 것으로 보인다. [4]에 따르면 128Mbit Direct RDRAM 의 경우 대기모드와 수면모드에서 활성모드로 진입하는데 소요되는 시간은 각각 2와 30사이클에 불과하며, 차단모드에서 활성모드로의 진입은 9,000사이클을 필요로 한다. 따라서 필요에 따라 힙 메모리를 대기, 수면, 차단 모드 등으로 비활성화시킴으로서 큰 성능저하 없이 에너지 절감이 가능할 것으로 판단된다.

6. 결 론

에너지 효율적인 JVM의 개발을 위해서는 자바 메모리에 대한 접근 형태를 분석하는 것이 무엇보다 중요하다. 임베디드 시스템의 경우 일반적으로 메모리 접근에 따른 동적 에너지 소비가 매우 큰 몫을 차지하고 있기 때문이다.

본 논문에서는 자바 응용 프로그램이 실행될 때 가장 빈번히 사용되는 바이트코드 명령들에 대해 자바가상기계 규격에 따라 그들의 메모리 접근 형태를 분석해 보았으며, 힙 메모리, 오퍼랜드 스택, 지역변수배열 등 논리적 메모리 구분에 따라 접근도를 조사하였다. 분석 및 실험 결과 오퍼랜드 스택이 가장 빈번히 사용되었으며, 거의 모든 바이트코드 실행마다 오퍼랜드 스택에 대한 접근이 이루어짐을 발견하였다. 즉 오퍼랜드 스택을 이루는 메모리를 특히 에너지 효율이 높은 기술로 구현해야 한다는 것이다. 읽기와 쓰기 비율은 6:4 정도로서 차이가 그다지 큰 편은 아니었다.

반면 힙 메모리는 가장 드물게 사용되었으며, 읽기와 쓰기 비율은 8:2 정도로 읽기 위주로 사용됨을 알 수 있었다. 힙에 대한 접근은 매우 불균일하게 일어났다. 즉 어느 시간에는 거의 사용되지 않다가 어느 순간에는 집중적으로 사용이 늘어나는 형태를 보였다. 이와같은 특징은 힙을 물리적 메모리 상에 구현할 때 대기, 수면, 차단, 불능 등 최근의 DRAM이 제공하고 있는 다양한 전력 모드를 활용함으로서 에너지 절감을 이룰 수 있음을 보여주는 것이다. 실제로 힙 메모리는 JVM에서 가장 큰 용량을 차지하고 있으며 모든 객체들과 배열이 저장되는 곳이므로 에너지 절감 효과가 더욱 클 것으로 예상된다. 지역변수배열은 힙과 마찬가지로 읽기 위주로 사용되었다. 읽기 동작은 오퍼랜드 스택과 마찬가지로 균일한 분포를 보였으며, 쓰기 동작은 다소의 불균일성을 나타내었다.

참 고 문 헌

[1] D. Kochnev and A. Terekhov, "Surviving Java for Mobiles", *IEEE Pervasive Computing*, April-June 2003, pp.90-95.

[2] G. Lawton, "Moving Java into Mobile Phones", *IEEE Computer*, Vol.35, No.6, June, 2002, pp.17-20.

[3] Yun Cao, Hiroyuki Tomiyama, Takanori Okuma, and Hiroto Yasuura, "Data Memory Design Considering Effective Bitwidth for Low-Energy Embedded Systems", *Proceeding of the 15th international symposium on System Synthesis (ISSS '02)*, 2002, pp.201-206.

[4] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "Memory Energy Management Using Software and Hardware Directed Power Mode Control", *TR: CSE-00-004*, Dept of Computer Sci and Engineering, Pennsylvania State University, 2000.

[5] J. Fryman, et al., "Energy-Efficient Network Memory for Ubiquitous Devices", *IEEE Micro*, Sept-Oct 2003, pp.60-70

[6] N. Vijaykrishnan, et al., "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower", *Proc. Int'l Symp on Computer Architecture*, June, 2000.

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

[8] 양희재, 자바가상기계, 한국학술정보, 2001, ISBN 89-5520-342-4.

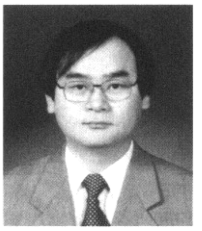
[9] G. Chen, M. Kandemir, N. Vijaykrishnan and W. Wolf, "Energy Savings Through Compression in Embedded Java Environment", *Proc. Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, 2002.

[10] D. Antonioli and M. Pilz, "Analysis of the Java Class File Format", Dept of Computer Sci., Univ of Zurich, *TR 98.4*, 1998.

[11] J. Waldron, "Dynamic Bytecode Usage by Object Oriented Java Programs", *TOOLS '99*, France, June, 1999.

[12] 양희재, "simpleRTJ 임베디드 자바가상기계의 ROMizer 분석 연구", 정보처리학회논문지 A, 제10-A권 5호, 2003. 10.

[13] *SPECjvm98 Benchmarks*, http://www.spec.org/osg/jvm98.



양 희 재

e-mail : hjyang@star.ks.ac.kr

1985년 부산대학교 전자공학과(공학사)

1987년 한국과학기술원 전기및전자공학과 (공학석사)

1991년 한국과학기술원 전기및전자공학과 (공학박사)

1991년~현재 경성대학교 컴퓨터공학과 교수

2001년~2002년 미국 펜실베이니아 주립대학교 교환교수

관심분야 : 임베디드 시스템, 유비쿼터스 컴퓨팅, 자바가상기계