자료 이동 측면에서 자바가상기계와 x86 프로세서의 비교

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

Comparison of Java Virtual Machine and x86 Processor in Data Transfer Viewpoint

Heejae Yang
Department of Computer Engineering
Kyungsung University

Abstract

This paper compares the differences between Java virtual machine and x86 processor in data transfer viewpoint. Memory models of JVM and x86 are analyzed and the data transfer paths are identified. As all operations must be performed to the values on operand stack, a great many data transfer operation is unavoidable in JVM. We also lists the number of data transfer operations necessary for executing some typical high-level language statements including assignment, arithmetic, conditional, and iterative statements.

I. 서론

자바 프로그램의 실행 환경인 자바가상기계(Java Virtual Machine)는 전형적 스택 기반 구조를 갖는 가상 컴퓨터다. JVM에서 모든 연산은 오퍼랜드 스택 상에 놓인 값들에 대해서만 이루어지며, 일체의 범용 목적 레지스터를 가정하지 않는다. 이런 특징으로 인해 JVM에서는 오퍼랜드 스택을 기준으로 한 자료 이동이 매우 빈번하게 일어난다. 관련 연구에 따르면 일반적인 자바 프로그램 실행 시 전체 실행된 명령어 중 40% 이상이 자료 이동을 위한 것들로 분석되었다 [1].

JVM은 범용 레지스터가 없으므로 모든 자료 이동은 메모리 상에서 일어나며, 메모리 상의 자료 이동은 시간지연과 더불어 에너지 소비를 일으키게 된다. 따라서 시간적으로 또한 에너지 사용면에서 효율적인 JVM 개발을 위해서는 자료 이동에 대한 연구가 필수적이다.

본 연구에서는 스택 기반 구조의 JVM과 전통적 레지스터 기반 구조를 갖는 컴퓨터를 자료 이동 측면에서 서로 비교해 보았다. 특히 본 연구에서는 PC 를 비롯해 여러 컴퓨터에서 광범위하게 사용되고 있는 x86 프로세서와 비교하였다.

JVM과 x86은 프로세서 구조 뿐 아니라 메모리 모델도 매우 다르다. JVM은 실행 시 메모리를 힙 메모리와 자바 스택 메모리로 나누고, 자바 스택 메모리는 다시 오퍼랜드 스택과 지역변수배열로 구성되는 메모리 모델을 사용하고 있다 [2]. 반면 x86 프로세서는 실행 시 메모리를 데이터 메모리와 스택 메모리로 나누고, 스택 메모리를 지역변수 영역으로 사용하는 모델을 사용한다 [3]. JVM에서는 메모리 영역간의 자료 이동을 위해 각각 전용 바이트코드 명령을 사용하고 있으나 x86에서는 `mov` 명령만으로 데이터 메모리, 스택 메모리, 레지스터 등 영역에 관계없이 자료 이동을 시

† 이 논문은 한국학술진흥재단 지역대학 우수과학자 지원에 의해 연구되었음 (R05-2004-000-10967-0).

킬 수 있다.

본 논문은 대표적인 프로그램 실행문, 즉 할당문, 산술문, 조건문, 반복문 등에 대해 JVM과 x86에서 각각 어떻게 자료 이동이 일어나고 있는지를 분석하였으며, 자료 이동에 따른 효율성을 서로 비교하였다. 본 연구의 결과는 효율적인 JVM의 개발에 큰 역할을 담당할 수 있을 것으로 기대된다.

II. 메모리 모델

2.1 JVM 의 메모리

JVM 의 메모리는 그림 1과 같이 클래스 영역, 힙 영역, 자바 스택 영역 등 세 곳으로 나뉘어진다. JVM 의 기계어에 해당되는 바이트코드 명령들은 클래스 영역에 배치되며, 각종 상수들은 이곳에 위치한 상수 풀(constant pool)에 저장된다.

힙은 객체들이 저장되는 곳이며, 객체들의 속성(attributes)인 필드(fields)들이 위치하게 된다. 배열도 이곳에 위치한다. 힙은 일반적으로 JVM에서 가장 큰 메모리를 사용하며, 더 이상 사용되지 않는 객체들에 대한 자동 쓰레기 수집(automatic garbage collection)이 일어나는 곳이기도 하다.

자바 스택에는 다수개의 스택 프레임이 놓인다. 스택 프레임은 메소드가 호출될 때 생성되는데, 현재 메소드가 다른 메소드를 호출하면 현재 스택 프레임 위에 또 다른 스택 프레임이 생기는 구조를 갖는다. 메소드가 끝나면 해당 스택 프레임도 사라진다.

스택 프레임의 내부는 그림 1의 오른 쪽에 보인 바와 같이 지역변수배열과 오퍼랜드 스택으로 구성된다. 지역변수배열(local variable array)은 메소드 호출 시 넘겨졌던 각종 인자(arguments)들이 놓이는 곳이며, 동시에 이 메소드 내에서만 사용되는 지역변수들이 위치하는 곳이기도 하다. 오퍼랜드 스택(operand stack)은 실제 연산이 일어나는 곳으로, 스택 기반 구조를 갖는 JVM 에서는 모든 연산이 이 오퍼랜드 스택 상에 있는 값들에 대해서 이루어진다. PC(program counter)는 현재 실행 중인 명령의 위치를 알려주며, SP(stack pointer)는 오퍼랜드 스택에서 제일 상위에 놓인 항목의 위치를 알려주는 용도로 사용된다.

2.2 x86 의 메모리

x86 의 메모리는 그림 2와 같이 코드 영역, 데이터 영역, 그리고 스택 영역 등으로 나뉘어진다. 비단 x86 뿐 아니라 거의 대부분의 실제 프로세서들이 이와 같은 메모리 모델을 따른다.

코드 영역은 x86의 기계어 명령들이 저장되며, 텍스트 구간(text segment)이라고 부르기도 한다. 데이터 영역은 초기화되어진 데이터들과 그렇지 않은 데이터들로 구분되어진다. 일반적으로 프로그램 내에서 사용되는 전역변수들이 이 영역에 저장된다.

스택 영역에는 마찬가지로 다수개의 스택 프레임이 놓인다. 스택 프레임은 함수가 호출될 때 생성되는데, 이 함수가 또다른 함수를 호출하면 현재 스택 프레임 상에 또 다른 스택 프레임이 생기는 구조를 갖는다.

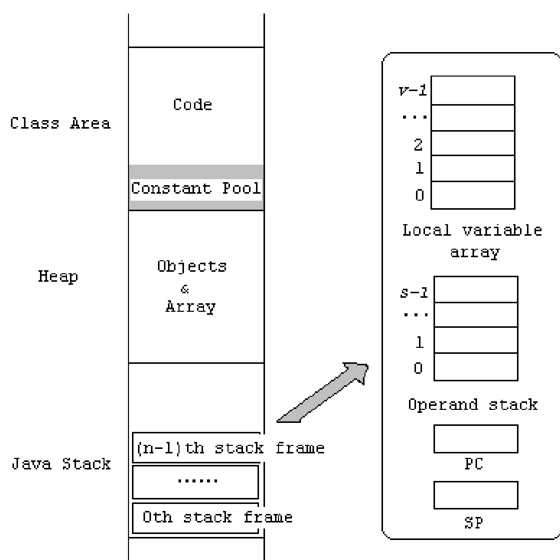

그림 1 JVM 메모리 모델

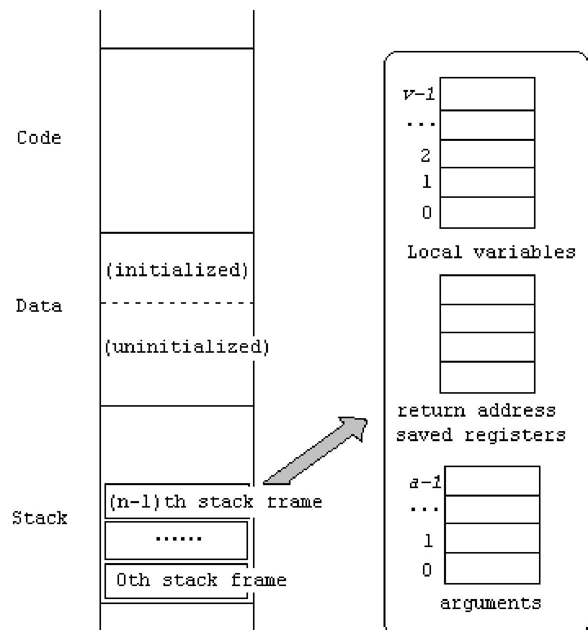

그림 2 x86 메모리 모델

함수가 끝나면 해당 스택 프레임도 사라지게 된다.

스택 프레임의 내부는 그림 2의 오른 쪽에 보인 바와 같이 지역변수, 복귀주소 및 기타 저장 레지스터, 함수로 넘겨진 인자 등을 저장하는 공간으로 구성된다. 지역변수(local variables)는 이 함수 내에서만 사용되는 지역변수들이 위치한다. 복귀주소 및 기타 저장 레지스터(return address & saved registers)에는 돌아갈 주소와 이 함수 내에서 변경되는 레지스터들이 저장된다. 인자(arguments) 부분에는 이 함수 호출 시 넘겨졌던 각종 인자들이 저장된다.

2.3 비교

JVM의 메모리 모델과 x86의 메모리 모델 사이에는 많은 유사점이 존재한다.

- 클래스 영역과 코드 영역은 명령어들의 집합체라는 점에서 동일하다.
- 힙 영역과 데이터 영역의 기능이 비슷하다. 힙에는 필드들이 들어가게 되는데, 필드는 해당 클래스 내의 메소드들에 대해 전역변수로 사용된다 [4]. 데이터 영역도 x86에서 전역변수들이 놓이는 곳이다.
- 자바 스택 영역과 스택 영역이 비슷하다. 둘 다 스택 프레임의 집합체로서, 메소드 또는 함수 호출 시 스택 프레임이 생기며, 메소드 또는 함수 리턴 시 사라진다. 스택 프레임에 들어있는 지역변수배열은 x86 스택 프레임의 지역변수 및 인자의 조합과 유사하다.

이상의 유사점 외에 차이점도 존재한다.

- 상수 모음이 JVM에서는 클래스 영역에 있지만(상수 풀) x86에서는 초기화 된 데이터 영역에 있다.
- 힙 영역의 메모리는 객체 생성 시 동적으로 할당되지만, 데이터 영역은 프로그램 시작부터 정적으로 존재한다.
- 힙 영역의 자료들은 자동 쓰레기 수집되지만, 데이터는 프로그램 일생동안 계속 유지된다.
- JVM에는 오퍼랜드 스택이 있지만 x86에는 없다.
- JVM에서 모든 연산은 오퍼랜드 스택에 있는 값에 대해 일어나지만, x86에서는 주로 레지스터에 있는 값들을 대상으로 일어나며, 데이터 영역이나 지역변수에 있는 값들도 종종 오퍼랜드가 된다.

III. 자료의 이동

3.1 JVM의 자료 이동

그림 3은 JVM의 자료 이동도를 나타낸 것이다. JVM에서 모든 연산은 오퍼랜드 스택의 값들에 대해 일어나므로 오퍼랜드 스택이 자료 이동의 중심이 되는 것은 당연하다.

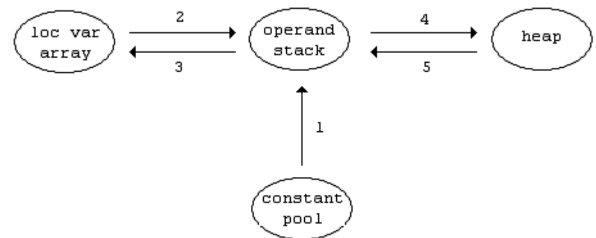

그림 3 JVM의 자료 이동경로

JVM은 각 경로별로 자료 이동을 위한 바이트코드를 따로 할당했다. 예를 들어 그림 3의 경로 1을 위해서는 `ldc` 명령을, 경로 2를 위해서는 `iload`, `aload` 등을, 경로 3을 위해서는 `istore`, `astore` 등의 명령을 사용하였으며, 경로 4와 5를 위해서는 각각 `putfield`, `getfield` 등의 명령을 할당하였다.

3.2 x86의 자료 이동

그림 4는 x86의 자료 이동도를 나타낸 것이다. 레지스터 기반 구조를 갖는 x86 프로세서에서는 거의 대부분의 연산이 레지스터 값들에 대해 일어나므로 레지스터가 자료 이동의 중심이 된다. JVM의 상수풀은 x86에서는 데이터 영역에 속하므로 그림 3의 경로 1은 그림 4의 경로 5에 포함된다.

x86에서는 자료 이동을 위한 명령을 각 경로별로 따로 할당하지 않고 `mov` 명령으로 통합하고 있다. 각 경로별 이동 명령의 예는 다음과 같다. `mov eax,ebx` (경로 1); `mov eax,[ebp+8]` (2); `mov [ebp+8],eax` (3); `mov [sum],eax` (4); `mov eax,[sum]` (5).

3.3 비교

JVM과 x86에서 공통적으로 발견되는 자료 이동의 유사점은 다음과 같다.

- 지역변수배열에서 지역변수배열, 지역변수에서 지역변수로의 자료 이동 경로는 존재하지 않는다.
- 힙에서 힙, 데이터에서 데이터로의 자료 이동 경로도 존재하지 않는다.
- JVM에서는 오퍼랜드 스택을 경유하여 지역변수 배열과 힙 사이의 자료 이동이 일어나며, x86에서는 레지스터를 경유하여 지역변수와 데이터 사이의 자료 이동이 일어난다.

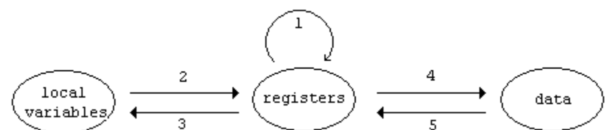

그림 4 x86 의 자료 이동 경로

또한 차이점도 다음과 같이 존재한다.

- x86에서는 레지스터에서 레지스터로의 자료 이동 경로가 있으며 활발히 사용된다. 반면 JVM에서 오퍼랜드 스택에서 오퍼랜드 스택으로의 자료 이동 경로는 존재하지 않는다 (dup, dup2, dup2_x 등이 있기는 하지만, 이동 경로라기보다는 값의 생성에 해당된다).
- JVM에서 오퍼랜드가 될 수 있는 값은 오퍼랜드 스택 상의 값밖에 없으므로 오퍼랜드 스택으로의 자료 이동이 매우 빈번하다. 반면 x86에서는 지역변수, 데이터, 레지스터 등이 모두 오퍼랜드가 될 수 있으므로 레지스터로의 자료 이동이 그다지 빈번하지 않다.

Ⅳ. 문장의 실행

고수준언어로 작성된 프로그램이 실행될 때 자료 이동이 어느 정도로 일어나는지 알기 위해 할당문, 산술문, 논리문, 조건문, 반복문 등 각 문장별로 분석해보았다. 문장은 JVM에서는 자바언어를, x86에서는 C언어를 각각 사용하였으며, J2SDK 1.4.1_04 자바 컴파일러와 gcc egcs-2.91.57 C 컴파일러를 각각 사용하여 번역했다. 문장 수행은 지역변수배열(JVM) 또는 지역변수(x86)를 오퍼랜드로 사용하는 것으로 가정했으며, 힙이나 데이터는 사용하지 않았다.

표 1은 각 문장 실행에 따른 자료 이동 회수를 정리한 것이다. 즉 각 문장을 바이트코드 또는 어셈블리어로 번역하여 순수하게 자료 이동 명령이 실행된 횟수를 조사하였다. 예상대로 JVM에서 x86에 비해 많은 자료 이동이 일어났으며, 주목할만한 점은 다음과 같다.

- 할당문에서 자료 이동은 JVM이나 x86이나 동일하다.
- 산술문, 조건문에서는 JVM에서의 자료 이동이 x86에서보다 2배 또는 3배 이상 일어난다.
- 가장 큰 차이는 반복문에서 발견된다. JVM에서는 반복 회수만큼의 자료 이동이 일어나지만, x86에서는 자료 이동이 전혀 없다.
- JVM의 지역변수배열 및 x86의 지역변수는 읽기 동작이 쓰기 동작보다 월등히 많이 일어난다.

서론에서 언급한 바와 같이 자료 이동은 시간 지연과 에너지 소비를 야기하게 된다. 표 1에서 알 수 있듯이 JVM은 x86 과 같은 일반 프로세서에 비해 많은 자료 이동을 필요로 하므로 속도지연 및 에너지 사용 증가라는 어려움을 겪을 수 밖에 없다. 즉 자바 프로그램이 일반 C 등 네이티브 프로그램에 비해 느린 까닭은 비단 인터프리터 방식을 사용하기 때문만 아니라 스택 기반 환경으로 인한 많은 자료 이동이 일어나기 때문이라는 해석이 가능하다.

표 1 문장 실행에 따른 자료 이동 회수

문장	예제코드	이동 회수			
		JVM		x86	
		경로2	경로3	경로2	경로3
할당문	i=j	1	1	1	1
산술문	i=i+j	2	1	1	0
조건문	if (i<10)	1	0	0	0
	if (i<j)	2	0	0	0
반복문	for (;i<10;i++)	10	0	0	0

Ⅴ. 결론

스택 기반 구조를 갖는 자바가상기계는 레지스터 기반 구조를 갖는 x86 등 범용 프로세서에 비해 많은 자료 이동을 필요로 한다. 본 논문에서는 JVM과 x86의 메모리 모델을 각각 비교하고 두 모델 사이의 유사점 및 차이점을 분석하였다. 또한 JVM과 x86의 자료 이동 경로를 각각 분석하였으며, 할당문, 산술문, 조건문, 반복문 등 전형적인 고수준 언어 프로그램 문장들이 실행될 때 몇 번의 자료 이동이 일어날 것인지 조사하였다.

분석 결과 JVM은 동일 문장 실행 시 2배 또는 3배 이상의 자료 이동을 필요로 하였으며, 특히 반복문 실행 시에는 반복 회수에 따라 자료 이동이 증가하는 것을 알 수 있었다. 이런 빈번한 자료 이동은 JVM 의 속도 저하 및 에너지의 과다 소비를 초래하므로 자료 이동을 최소화 할 수 있는 새로운 구조 개발 및 프로그래밍 기법의 개발이 필요하다고 하겠다.

참고문헌

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

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

[3] John Uffenbeck, *The 80x86 Family: Design, Programming, and Interfacing*, 2nd ed., Prentice Hall, 1998

[4] 양희재, "전형적 자바 프로그램에서 필드의 사용 형태", 한국정보과학회 제32회 추계학술대회, 2005. 11.