자바 반복문의 에너지 효율성

양희재

경성대학교 컴퓨터공학과

Energy Efficiency of Iteration Statement in Java

Heejae Yang

Department of Computer Engineering, Kyungsung University

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

요 약

자바 프로그램의 실행 환경인 자바가상기계에서는 거의 모든 바이트코드들이 메모리 상에서의 자료 이동을 필요로 한다. 자료 이동은 에너지 소비를 일으키므로 이것의 최소화는 JVM의 에너지 효율성 향상에 매우 중요하다. 특히 자바 반복문은 반복 회수에 비례하는 양만큼의 자료 이동을 요구하므로 JVM의 에너지 효율을 높이기 위해서는 무엇보다도 자바 반복문의 에너지 효율을 높이는 것이 중요하다. 본 논문에서는 자바 반복문을 바이트코드 수준에서 분석하여 에너지 효율성을 높일 수 있는 여러 방법들을 제안하였다.

ABSTRACT

In Java virtual machine which is the running environment of Java programs, almost every bytecode execution requires data transfers in memory. Data transfer incurs energy consumption and hence minimizing the transfer operation is very important for improving the energy efficiency of JVM. As the number of data transfers for a Java iterative statement is directly proportional to the iteration count, improving the energy efficiency of iterative statement is crucial to keep the energy efficiency of JVM high. This paper analyzes Java iterative statement at bytecode level and proposes some points how to improve the energy efficiency.

키워드

Java, Java virtual machine, low-power system, bytecode

Ⅰ 서 론

스택 기반의 구조를 갖는 자바가상기계(Java Virtual Machine)에서는 모든 연산이 스택 상에 놓인 오퍼랜드들에 대해 이루어지며, 별도의 범용 레지스터는 존재하지 않는다. 따라서 대부분의 문장 수행 시 오퍼랜드 스택을 중심으로 하는 수많은 자료 이동이 요구되어진다. 자료 이동은 메모리 접근을 필요로 하며, 메모리 접근은 에너지 소비와 밀접한 관련성을 갖는다. 즉 에너지 효율성을 높이기 위해서는 자료 이동을 최소화시켜야 한다.

자바 바이트코드의 분석 결과 자바 프로그램에서 반복문이 가장 많은 자료 이동을 필요로 하는 것을 발견할 수 있었다. 즉 할당문, 산술문, 조건문 등은 대부분 4회 이내의 자료 이동을 요구하지만, 반복문은 반복 회수에 비례하는 만큼의 자료 이동을 요구한다 [1]. 따라서 JVM의 에너지 효율성을 높이기 위해서는 무엇보다도 반복문이 필요로 하는 에너지 소비를 줄일 수 있어야 한다.

본 논문에서는 자바 반복문을 바이트코드 수준에서 분석하였다. 반복이 이루어질 때마다 출구조건을 확인하기 위한 비교 동작이 이루어지는데, 비교 대상이 상수인지 지역변수인지 또는 필드인지에 따라 각기 다른 수준의 에너지가 소비되어진다. 본 연구에서는 비교 대상들에 따른 에너지 소비를 각각 분석해보고 효율 향상을 위한 방법들을 제안하였다.

본 논문의 구성은 다음과 같다. 2장에서는 자료들이 위치하는 JVM 메모리에 대해 설명하며, 3장에서 널리 쓰이는 for 반복문에 대해 에너지

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

효율성을 바이트코드 수준에서 해석한다. 분석 내용을 4장에서 실험을 통해 확인하였으며, 5장에서 결론을 내린다.

II 자바 메모리

바이트코드의 오퍼랜드로 사용되는 자바 메모리는 상수풀 (constant pool), 힙 (heap), 그리고 자바 스택 (Java stack) 등 세가지로 나눌 수 있다. 상수풀에는 16비트 이상으로 표현되는 정수, 실수, 문자열 등이 들어있으며, 힙에는 각 객체들의 필드들이 저장된다. 자바 스택은 하나의 메소드가 호출될 때마다 한 개씩 생성되며, 메소드가 반환되면 사라진다. 자바 스택의 내부는 또다시 두 개의 메모리 영역으로 나눌 수 있는데, 지역변수들을 저장하는 지역변수배열(local variable array)과 실제 연산이 일어나는 오퍼랜드 스택 (operand stack) 등이 여기에 해당된다. 바이트코드가 실행되면서 상수풀, 지역변수배열, 오퍼랜드 스택, 힙 등 메모리 사이에 활발한 자료이동이 이루어진다 [2].

III 반복문의 해석

자바 프로그램에서 제공되는 반복문의 종류로는 for 문, while 문, do-while 문 등이 있지만, 본질적으로 큰 차이는 없다. 지면 제한 등으로 인해 본 논문에서는 for를 사용하는 반복문으로 한정하였지만, 다른 반복문 형식도 동일한 기준으로 적용할 수 있다.

3.1 for 문의 형식

for 문의 일반적 형식은 다음과 같다 [3].

for (*init*; *expression*; *update*) *statement*

이 형식을 보면 for 문과 관련된 요소로는 다음 네 가지가 있음을 알 수 있다.

▷ *init*: 초기화 문장이며 단 1회 실행된다.
▷ *expression*: 참/거짓값(true/false)을 반환하는 표현식으로서 출구조건에 해당된다. 산출된 값이 참이면 반복을 계속하고, 거짓이면 반복이 끝난다.
▷ *statement*: 실제 반복적으로 실행되어지는 문장이다.
▷ *update*: 반복이 일어난 후 반복제어변수값을 새로운 값으로 바꾼다.

for 문에서 *init* 는 단 1회 실행되지만, 나머지 부분은 모두 반복회수만큼 실행된다. 따라서 *init* 을 제외한 나머지 부분은 매우 효율적 코드로 작성되어야만 에너지 효율성을 높일 수 있다. 본 논문에서 의미하는 효율적 코드란 자료 이동을 최소화할 수 있는 코드를 의미한다.

위 문장에서 *statement* 는 반복 수행하고자 하는 목적에 따라 각각 다르게 구현된다. 간단하게 하나의 문장으로 표현되는 경우도 있을 것이며, 매우 복잡하게 수십 개, 또는 수백 개 이상의 문장으로 구성되는 경우도 있다.

즉 *statement* 부분은 그 종류가 너무 많기 때문에 효율성을 논하기에 적합치 않으며, *init* 부분은 단 1회 실행되기 때문에 효율과는 상관이 적다. 따라서 본 연구의 관심은 *expression* 과 *update* 의 효율성에 있다.

for 문에서 반복 회수를 결정하는 것은 반복제어변수이다. 반복제어변수(loop control variable)는 말 그대로 변수이므로 자바 프로그램에서는 지역변수 또는 필드를 사용하여 이 변수를 구현할 수 있다. 일반적인 관례는 지역변수를 사용하는 것이다. 대체적으로 반복제어변수는 정수형이며 반복될 때마다 1만큼 증가한다.

아래 프로그램은 대표적인 for 문의 형태를 보여주고 있다.

```
for (int i=0; i<100; i++)
        ... statement ...
```

위 프로그램을 바이트코드로 번역하면 그림 1과 같다.

```
        iconst_0            ; init
        istore_1
        goto compare
loop:
        ......              ; statement
        iinc 1 1            ; update
compare:
        iload_1             ; expression
        bipush 100
        if_icmplt loop
done:
```

그림 1 for 문의 바이트코드 번역

이제 본 논문의 관심 부분인 *expression*과 *update*에 대해 각각 고찰해보자.

3.2 expression

for 문에서 *expression* 은 출구조건을 위해 필요하며, *expression* 자체는 진리값을 반환한다. 그림 1의 예와 같이 대부분의 for 문은 반복제어변수와 다른 값을 서로 비교하는 *expression* 을 사용한다. 이때 다른 값은 상수, 다른 지역변수, 필드 등이 있을 수 있다.

(1) 상수와의 비교

먼저 상수와의 비교에 대해 알아보자. 그림 1에서 사용한 상수값은 100인데, 이 값을 반복제어변수와 비교하기 위해 오퍼랜드 스택에 넣는 것

```
for (int i=99; i>=0; i--)
                ... statement ...

        bipush 99           ; init
        istore_1
        goto compare
loop:
        ......              ; statement
        iinc 1 -1           ; update
compare:
        iload_1             ; expression
        ifge loop
done:
```

그림 2 0과 비교한 반복문

을 볼 수 있다.

만일 반복제어변수와 비교할 상수값이 0이라면 굳이 상수를 오퍼랜드 스택에 넣을 필요가 없다. JVM은 어떤 수를 0과 비교하기 위해 일련의 바이트코드를 제공하고 있기 때문이다. 그림 1의 문장을 0과 비교하는 프로그램으로 바꾸면 바이트코드 번역은 그림 2와 같아진다. 그림 2에서는 그림 1과 동일하게 100번을 반복하지만 *expression* 부분의 코드 길이가 하나 줄어듦에 주목하여라. 즉 *expression* 을 작성할 때 0과 비교하는 편이 효율이 높다는 의미이다.

상수값이 8비트로 나타내어질 수 있다면 bipush 가, 16비트로 나타내어질 수 있다면 sipush 가 사용된다. 그 이상의 크기를 갖는 상수는 상수 풀에 있으므로 ldc 명령을 사용하여 상수 풀에서 오퍼랜드 스택으로의 자료 이동이 1회 발생하게 된다. 즉 *expression* 을 작성할 때 가능한 절대값이 작은 값과 비교하는 편이 효율이 높다는 의미이다. 결론적으로 상수와 비교하되, 0이 가장 좋고, 그렇지 않으면 8비트나 16비트 상수와 비교한다. 32비트 상수와의 비교는 효율면에서 좋지 않다.

(2) 지역변수와의 비교

반복제어변수와 비교할 값이 상수가 아니라 다른 지역변수라면 오퍼랜드 스택에는 반복제어변수를 포함하여 2개의 지역변수가 들어가게 된다. 즉 지역변수배열에서 오퍼랜드 스택으로 2회의 자료 이동이 일어난다는 것이다. 상수와의 비교에 비해 효율이 낮아지게 된다.

(3) 필드와의 비교

반복제어변수와 비교할 값이 필드라면 오퍼랜드 스택에는 반복제어변수 외에도 필드값이 들어가야 한다. 필드값을 가져오기 위해서는 0번째 지역변수배열을 먼저 오퍼랜드 스택에 가져오고, 다음에 필드를 가져올 수 있기 때문에 전체적으로는 지역변수배열에서 오퍼랜드 스택으로 2회의 자료 이동, 필드가 들어있는 힙 메모리에서 오퍼랜드 스택으로 1회의 자료 이동 등 모두 3회의 자료 이동이 일어난다.

만일 반복제어변수도 지역변수가 아니라 필드에 있으며, 비교할 값도 필드에 있다면 효율은 더욱 낮아진다. 두개의 필드값을 오퍼랜드 스택으로 옮겨야 하므로 지역변수배열에서 오퍼랜드 스택으로 2회, 힙 메모리에서 오퍼랜드 스택으로 2회 등 총 4회의 자료 이동이 발생하게 된다.

(4) 정리

표 1은 *expression* 의 비교 대상에 따른 자료 이동 회수를 정리한 것이다. 세로항목을 가로항목과 비교하는 것이며, 가장 효율이 좋은 조합은 지역변수와 상수를 비교하는 것이고 가장 효율이 낮은 조합은 필드와 필드를 비교하는 것임을 알 수 있다.

표 1 비교 대상에 따른 자료 이동 회수

	const	const32	loc	field
loc	1	2	2	3
field	2	3	3	4

표1에서 *expression* 내에 필드가 들어가면 효율이 낮아지는 것을 볼 수 있다. 따라서 필요에 따라 필드값을 일단 지역변수로 복사하고, 이후 비교는 지역변수와 하도록 하는 것이 더 효율적이라고 볼 수 있다.

또한 *expression* 은 반복실행 되므로 *expression* 내에 메소드 등을 두는 것은 효율을 떨어뜨린다. 매 반복시마다 메소드의 반환값이 달라지면 할 수 없지만, 그렇지 않다면 메소드 반환값을 지역변수에 저장하고, 이후 비교는 이 지역변수와 하는 것이 더 효율적이다.

3.3 update

일반적으로 for 문이 반복될 때마다 반복제어변수값이 1만큼 증가한다. JVM은 iinc 라는 바이트코드를 제공하는데, 이것의 형식은 다음과 같다.

```
iinc i j
```

이 명령의 의미는 i번째 지역변수배열 항목을 j 만큼 증가시키라는 것이다. 이 명령은 모든 연산이 지역변수 내에서만 일어나므로 다른 영역 메모리로의 자료 이동은 없는 것으로 간주할 수 있다.

스택 기반 구조를 갖는 JVM에서 모든 연산은 오퍼랜드 스택 내에서만 이루어지는데, 오퍼랜드 스택이 아닌 지역변수배열에서 일어나는 유일한 동작이 바로 iinc 이다. 만일 iinc 라는 명령이 제공되지 않는다면 동일한 기능을 하기 위해 다음과 같은 명령들이 필요하다.

```
iload_i
bipush j
```

```
add
istore_i
```

즉 iinc 명령이 없다면 위와 같이 4개의 바이트코드가 필요하며, 자료 이동도 2회나 더 일어난다. 시간적 지연과 에너지 낭비도 발생하여 효율이 낮아질 수 밖에 없다.

update 는 자바 반복문에서 반복 회수만큼 일어나므로 그 영향은 매우 크다. JVM 설계자가 처음부터 iinc 를 반복제어변수의 업데이트를 위한 목적으로 포함시켰는지 알 수 없으나 iinc 는 JVM의 효율을 높이는 중요한 명령이 되었다.

IV 실험 및 분석

3장에서 분석한 내용들을 확인하기 위해 실제 JVM 상에서 실험을 하였다. 사용한 JVM은 썬마이크로시스템사의 HotSpot Client VM (1.4.1_04 판)이며, 이 환경이 제공하는 자바 컴파일러를 사용하여 바이트코드 번역을 하였다. 바이트코드 수준에서의 동작만을 보기 위해 JVM은 항상 인터프리터 모드에서만 실행되도록 설정했다. 실험환경은 펜티엄 2.4GHz 프로세서, 256MB 메모리를 가지며, Windows XP Professional 운영체제 (2002년판)를 사용하는 컴퓨터이다.

먼저 for 문의 *expression*에서 비교 대상을 지역변수와 상수로 한 경우를 분석했다. 그림 3은 반복회수(count)를 10,000번부터 100,000번까지 바꾸었을 때 반복이 끝날 때까지 소요된 시간(msec)을 나타낸 것이다. 이 그림에서 반복회수가 30,000번부터 40,000번 사이에 그래프 상에 변화가 일어나는 것을 발견할 수 있다. 즉 반복회수가 16비트 최대수인 32,767까지는 bipush, sipush 등의 바이트코드가 사용되지만, 32,768부터는 ldc 명령에 의해 상수풀에 대한 접근이 추가되므로 시간의 증가가 발생한 것이다.

지면의 제약으로 인해 이하 실험에 대해서는 일부 경우에 대해서만 설명하고자 한다. *expression*에서 0이 아닌 상수를 사용했을 때와 0을 사용했을 때 소요된 시간을 비교해보면 반복회수가 10,000번과 50,000번에서 각각 4,109/3,547, 25,797/17,750으로 나타났는데, 이것은 0을 사용한 *expression* 이 훨씬 효율적임을 보여주는 것이다.

또한 *expression*에서 비교대상을 지역변수/지역변수, 지역변수/필드, 필드/필드로 각각 달리 했을 때 반복회수가 10,000번인 경우 각각 4,078, 4,360, 8,859였으며, 50,000번인 경우 20,391, 21,843, 44,063으로 나타났다. 지역변수/지역변수의 비교에 비해 필드/필드의 비교는 2배 이상 많은 에너지 소비를 필요로 한다는 것을 알 수 있다.

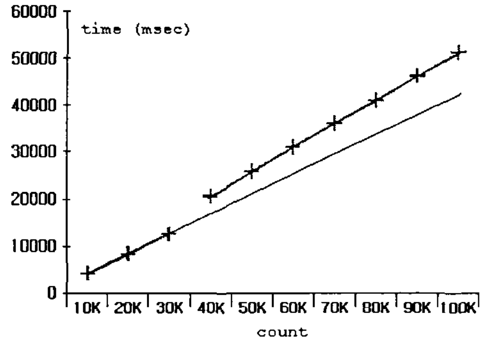

그림 3 반복회수에 따른 실행 시간 변동

V 결 론

본 논문에서는 자바 반복문을 바이트코드 수준에서 분석하여 이 문장이 실행될 때 소비되는 에너지를 분석해보았다. 반복문은 출구조건을 찾기 위해 필연적으로 값을 비교하게 되는데, 비교하는 값이 상수, 지역변수, 필드인 경우 각기 다른 에너지를 소비한다. 소비되는 에너지를 정량적으로 분석하였으며, 실험을 통해 확인하였다. 본 논문의 결과는 에너지 및 시간면에서 효율적인 자바 프로그래밍 작성에 큰 도움을 줄 것으로 기대된다.

참고문헌

[1] 양희재, "자바 문장 형식이 프로그램 실행시간에 미치는 영향," 제24회 한국정보처리학회 학술대회, 2005. 11.

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

[3] J. Gosling, et al., *The Java Language Specification*, 3rd ed., Sun Microsystems Press, 2005