제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

루프 바이트코드의 정의를 통한 자바가상머신의 성능 개선

이지현*, 원희선**, 문경덕**, 김영국*
*충남대 컴퓨터 과학과
** 한국전자통신연구원 정보가전연구부
e-mail : {jhlee, ykim}@cs.cnu.ac.kr/{ hswon,kdmoon}@etri.re.kr

A Definition of Loop Byteocode for Performance Improvement of Java Virtual Machine

Ji-Hyun Lee*, Hee-sun Won**, Kyung-Doek Moon**, Young-Kuk Kim*
* Dept. of Computer Science, Chungnam National University
** Information Appliance Technical Department, ETRI

요 약

자바가상머신은 플랫폼에 독립적인 실행을 위해서 바이트코드라고 하는 스택(stack) 기반의 가상 기계어를 사용하므로 실행 속도가 느리다는 단점이 있다. 특히 루프문을 포함하는 자바프로그램을 자바가상머신에서 수행 시키면 루프에 관련된 몇 개의 동일한 바이트코드가 루프의 실행 횟수만큼 반복적으로 인터프리트해서 수행하므로 상당한 성능 저하를 유발한다. 본 논문에서는 이런 비효율적인 성능상의 문제점을 개선하기 위해 루프를 수행하는 새로운 바이트코드를 정의 및 구현하고, 이를 실제 클래스 파일에 적용하기 위한 코드 변경 절차와 방법을 제시한다. 제안된 바이트코드를 사용해서 루프의 처리 속도를 개선할 경우, 클래스 파일의 크기를 줄일 수 있을 뿐만 아니라 간단한 성능 평가를 통해서 자바가상머신의 성능 개선 효과를 확인할 수 있다.

1. 서론

자바 바이트코드는 다양한 이기종 환경에서 독립적인 실행을 위해 자바 컴파일러에 의해서 생성되는 가상 기계 코드이다. 자바 바이트코드는 플랫폼에 독립적으로 실행될 수 있는 장점이 있지만, 스택을 기반으로 하는 가상기계 코드이고 인터프리팅을 통해 실행되는 특징으로 다른 언어에 비해 실행 속도를 현저히 느리게 하는 단점이 있다.

특히 루프문을 포함한 자바프로그램을 자바가상머신에서 수행할 경우 루프의 제어변수 초기화, 저장, 적재, 분기, 제어변수의 증가 또는 감소, 비교 등과 같이 루프에 관련된 몇 개의 동일한 바이트코드를 인터프리트해서 스택 기반의 연산을 루프의 실행 횟수만큼 반복적으로 수행하므로 자바가상머신의 성능을 크게 저하시키는 결과를 초래할 수 있다.

중첩된 루프를 수행하는 경우에는 외곽 루프(outer loop)가 한번 수행될 때마다 내부 루프(inner loop)의 전체가 매번 동일하게 수행되므로 많은 오버헤드가 발생한다. 따라서 루프의 실행 횟수가 커질수록, 중첩되는 루프의 수가 많아질수록 전체적인 성능은 더욱 나빠진다.

본 논문에서는 자바가상머신에서 루프문을 수행할 경우에 동일한 바이트코드가 반복적으로 수행되는 비효율성을 개선하기 위해서 자바가상머신에서 아직 정의하지 않은 새로운 바이트코드를 제안한다. 또한 제안된 바이트코드를 클래스 파일에 적용해 자바가상머신에서 수행시킨다. 이 방법은 현재 실행되는 시스템에 맞는 기계어 코드로의 변환을 통해 성능을 개선하는 플랫폼에 의존적인 방법과는 달리 자바가상머신 내부에서 새롭게 정의된 바이트코드의 추가를 통해서 수행되므로 플랫폼에 독립적이다. 따라서 이런 새로운 바이트코드를 추가해서 수행한 경우에 어느 정도의 성능 개선이 되었는지 알아보기 위해 간단한 성능 분석을 하였다.

본 논문의 구성은 다음과 같다. 2 장에서는 클래스 파일과 바이트코드의 확장에 대해서 알아본다. 3 장에서는 루프문을 실행하는 과정에 대해서 설명한 뒤 본

논문에서 제안한 새로 정의된 루프 명령어와 중첩된 루프의 처리 방법에 대해서 기술한다. 4 장에서는 새로 정의된 바이트코드를 실제적으로 구현하여 어느 정도의 성능 향상이 이루어졌는지 분석하며 마지막으로 5 장에서는 결론과 향후 연구방향에 대해서 기술한다.

2. 관련연구

2.1 자바 클래스 파일

자바 소스 프로그램은 자바 컴파일러를 통하여 클래스 파일로 생성된다. 바이트스트림인 클래스 파일 즉, 바이트코드는 8 비트 단위로 읽혀져 해당 플랫폼에 알맞은 형태로 번역된다.

자바 바이트코드는 전형적인 어셈블리 언어 스타일을 갖는다. 바이트코드는 스택 기반 가상 기계어로서 스택에 대한 동작들을 정의하고 오퍼랜드 스택으로부터 오퍼랜드를 가져온다. 많은 명령어는 오퍼랜드 스택으로부터 값을 팝하고 연산하여 그 결과를 푸쉬한다. 또한 자바가상머신은 임의의 변수를 저장하기 위한 레지스터가 없기 때문에 로컬 변수를 갖으며 명령어는 인덱스에 의해서 참조되는 레지스터로 로컬 변수를 취급한다.[1]

바이트코드는 인터프리트되기 쉬운 장점을 갖고 있지만 자바가상머신이 스택 기반 이므로 연산을 일련화하고, 값을 재사용할 수 없기 때문에 스택과 로컬 변수에 대한 불필요한 적재(Load)와 저장(Store)에 관한 명령어가 많아진다는 단점을 가지고 있다.[5]

2.2 바이트코드의 확장

자바가상머신은 200 개의 명령어로 구성되어 있으며 자바가상머신 내부에서 사용되기 위해서 예약(reserved)되어있는 3 개의 명령어가 있다. 그리고 썬사에서 인터프리터 되어진 바이트코드의 성능 향상을 위해 25 개의 "_quick" 명령어를 자바가상머신에 내부적으로 구현하였다. 3 개의 예약된 명령어와는 다르게 자바가상머신에서는 썬사의 "_quick" 명령어처럼 새로운 기능을 하는 명령어들을 더 추가할 수 있도록 하였다.[2]

썬사의 "_quick" 명령어는 이미 해석(resolved)되어 있는 컨스턴트 풀 엔트리(Constant Pool Entry)를 참조하는 바이트코드를 수행하는 경우에 속도향상을 위해서 고안된 바이트코드이다. 즉, 자바가상머신에서 컨스턴트 풀 엔트리가 이미 해석되어져 있다면 이것을 참조하는 명령어를 "_quick" 명령어로 변경해 준다. 예를 들어, ldc 명령어가 이미 해석되어 있는 컨스턴트 풀 엔트리를 참조한다면 ldc 명령어를 "_quick"명령어로 바이트 스트림에서 바꾸어준다.

3. 자바가상머신의 루프 성능 개선

3.1 루프문의 실행 방법

자바 프로그램에서 루프문은 제어변수의 초기값을 설정해주고 조건이 만족할 때까지 제한 변수의 값을 증가 또는 감소 시키면서 루프문 안에 있는 프로그램 코드를 실행 시킨다.[3][4] 다음은 루프를 100 번 실행하는 자바 프로그램과 바이트코드이다.

```
Public static void main(String args[]) {
        For(int i =0; I<100; I++) {
        [ 루프문안에 수행되는 프로그램 ]
        }
}
```

[그림 1] 자바프로그램에서 루프문

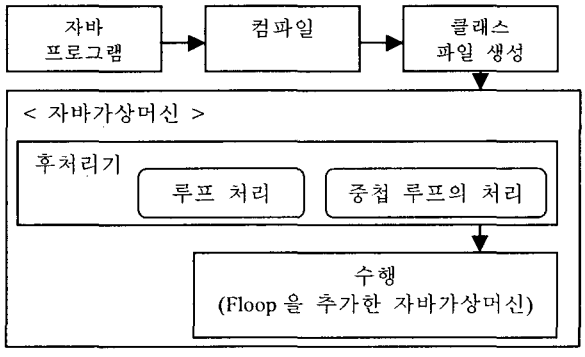
PC(program counter)	bytecode
0	**iconst_0 //제어변수의 초기화**
1	**istore_1 //제어변수저장**
2	**goto 8 //분기**
[루프문안에서 수행되는 바이트코드들]	
5	**iinc 1 1 // 제어변수 증가**
8	**iload_1 // 제어변수 적재**
9	**bipush 100 // 조건값 저장**
11	**if_icmplt 5 // 비교 & 분기**
14	return

[그림 2] 루프를 수행하는 바이트코드

[그림 2]는 [그림 1]의 루프문을 자바가상머신에서 수행하는 바이트코드이다. 위의 바이트코드들을 수행하면 루프의 실행 횟수인 100 번 만큼 프로그램 카운터 0-2 그리고 5-11 까지 반복된다. 이런 경우에 루프문에 관련된 제어변수의 초기화와 저장, 분기, 증가 또는 감소, 제어변수의 적재, 비교를 반복적으로 수행하므로 오버헤드가 크다. 또한 자바가상머신에서 수행될 때 스택과 로컬 변수에 대한 연산이 빈번하게 이루어진다. 따라서 자바프로그램에서 루프의 실행 횟수가 많거나 중첩된 루프가 많은 경우에는 동일한 바이트코드들이 반복적으로 수행되므로 자바가상머신의 성능에 많은 영향을 미칠 수 있다.

3.1 루프의 성능 개선 방법

본 논문에서는 자바가상머신에서 수행되는 루프의 성능 개선을 위하여 새로운 바이트코드인 Floop 을 정의하였다. 이를 실제 클래스 파일에 적용해서 수행되는 개략적인 모습을 [그림 3]에서 보여주고 있다. 이때 자바가상머신은 Floop 바이트코드에 대한 기능을 정의해주어야 한다.



[그림 3] 루프 바이트코드를 확장한 시스템의 개요

위의 [그림 3]에서 보여주듯이 자바프로그램을 컴파일해서 생성된 클래스 파일을 수행하기 전에 후처리기에서 받아들인다. 후처리기는 클래스 파일을 읽어들여서 루프 또는 중첩된 루프에 관련된 바이트코드들이 수행되는지 검사한다. 루프가 수행되는 것이 발견되면 Floop 바이트코드와 오퍼랜드(operand)로 변경해 준다. 만약 루프문 안에서 수행되는 바이트코드 중에서 제어변수의 값을 변경하는 경우는 이번 구현에서 제외하였다. 아래의 [그림 4]는 후처리기에서 루프와 중첩된 루프를 검사하는 알고리즘이다.

< 알고리즘 >

1. 루프의 제어 변수의 초기화를 위해 스택에 있는 값을 푸쉬하여 로컬 변수에 값을 저장하는 바이트코드와 goto 명령어가 있는지 검사한다.
2. goto 명령어의 분기 번지로 점프하여 제어 변수의 값을 로컬 변수에 적재하고 조건문에서 비교되는 값을 가져오는 바이트코드가 있는지 검사한다. 위의 두 조건을 만족하면 두 개의 변수를 비교하여 분기하는 바이트코드가 있는지 검사한다.
3. 2 번에서 분기되는 번지로 프로그램 카운터를 옮긴 후 루프문안에 있는 블록들을 수행한다. 블록들이 모두 수행된 후에 제어변수가 증가 또는 감소되는지 검사한다.
4. 위의 조건을 모두 만족하면 루프문을 포함한 프로그램이 수행되는 바이트코드이므로 Floop 명령어와 오퍼랜드들로 변경한다.
5. 루프문 안에서 수행되는 블록에서 또 다른 루프문을 수행하는 바이트코드가 있는지, 즉 중첩된 루프가 있는지 1 번에서부터 4 번까지의 과정을 거쳐서 검사한다. 만약 중첩된 루프이면 Floop 명령어로 바꾸어주고 다시 한번 내부 루프문 안에서 수행되는 블록중에 루프문을 수행하는 바이트코드가 있는지 검사한다.
6. 이렇게 중첩된 루프가 있는지 계속 검사를 해주며 만약 루프에 관련된 바이트코드가 없으면 후처리기는 종료된다.

[그림 4] 후처리기에 적용되는 알고리즘

자바가상머신에서 이렇게 변경된 클래스 파일을 수행하기 위해서 *Floop* 의 기능을 정의할 때 제어변수의 초기값, 제어변수의 증가 또는 감소값, 조건식의 경계값 등을 오퍼랜드로서 제공해야 한다. [그림 5]은 Floop 바이트코드와 오퍼랜드에 대해서 설명하였다.

* 바이트코드 : Floop
* 오퍼랜드 : *init, increment, bound, local_index*
- init : 제어 변수의 초기값
- increment : 제어 변수의 증가 또는 감소 값
- bound : 루프문에서 조건식의 경계 값
- local_index : 제어 변수의 로컬 인덱스 값

[그림 5] Floop 의 명령어와 오퍼랜드의 형식

따라서 이런 Floop 바이트코드를 [그림 2]의 바이트코드에 적용한 결과를 [그림 6]에서 보여주고 있다. 새로운 바이트코드의 추가로 인하여 클래스 파일의 크기도 줄어들었으며 동일한 바이트코드들이 반복적으로 수행되는 비효율적인 면을 개선할 수 있다.

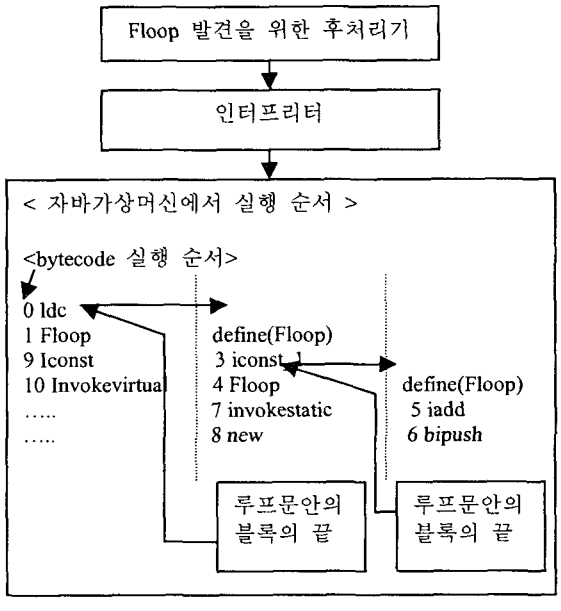
PC(program counter)	bytecode
0	**Floop 1, 0, 1, 100**
[루프문안에서 수행되는 바이트코드들]	
14	return

[그림 6] Floop 으로 변경한 바이트코드

3.2 중첩된 루프의 실행

중첩된 루프란 루프 속에 다른 루프가 있는 경우를 말한다. 중첩된 루프에서 내부 루프(inner loop)는 외곽 루프(outer loop)가 한번 실행될 때마다 전체를 반복 실행하므로 외곽 루프가 매번 실행될 때마다 내부 루프는 똑같은 일을 한다. 자바가상머신에서 스택 기반의 바이트코드를 인터프리팅해서 중첩된 루프를 수행하면 단일 루프를 수행하는 경우보다 더 많은 오버헤드가 발생된다.

따라서 중첩된 루프의 비효율적인 수행을 개선하기 위해서 Floop 바이트코드를 이용한다. [그림 3]의 후처리기에서 중첩된 루프를 Floop 바이트코드로 변경한 뒤 변경된 중첩된 루프가 자바가상머신에서 수행되어지는 모습을 [그림 7]에서 보여주고 있다.



[그림 7] 중첩된 루프의 실행 과정

위에서는 3 개의 중첩된 루프를 Floop 로 변경하여 수행되는 과정을 보여주고 있다. 각각의 루프는 자신의 루프문안에 있는 블록들을 수행한다. 이때 수행되는 블록들 중에 다시 Floop 바이트코드가 수행되는 경우에는 내부 루프문 안에 있는 블록을 먼저 실행해준

다. 다시 외곽 루프로 제어가 돌아오면 외곽루프는 자신의 제어 변수를 증가 또는 감소 시킨 후에 다시 내부 루프를 수행시킨다. 이렇게 내부 루프문 안에 있는 블록들이 다 수행되고 외곽 루프의 블록들도 다 수행되면 프로그램은 종료하게 된다.

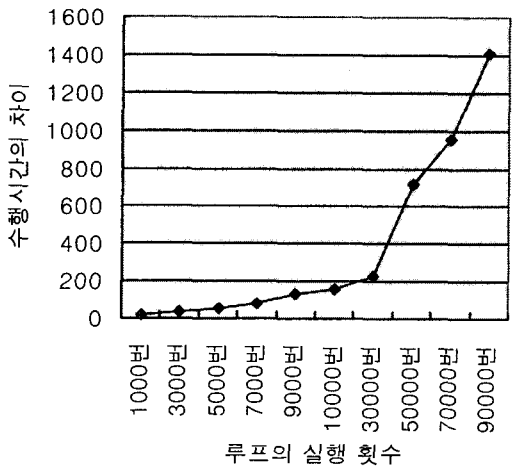
이런 중첩된 루프인 경우에는 바이트코드도 많이 줄어들었을 뿐만 아니라 외곽 루프를 실행할 때 내부 루프를 매번 동일하게 수행되는 것을 개선하였다. 따라서 자바가상머신에서 수행 시에 단일 루프에서 얻을 수 있는 성능 개선의 효과보다 더 많은 성능 효과를 기대할 수 있다.

4. 구현 및 성능 평가

실제적인 Floop 바이트코드의 구현과 성능 분석을 위해서 레드햇 리눅스 7.1 에서 인터프리터 방식으로 수행되는 자바가상머신 kaffe 1.0.6 을 테스트 기반으로 하였다.

테스트 프로그램은 루프와 중첩된 루프를 많이 사용하는 애플리케이션을 대상으로 하였으며 성능 측정은 자바가상머신에서 루프를 수행할 때 리눅스 시간으로 측정하였다. 루프의 실행 횟수를 증가시키면서 기존의 수행 방법과 새로운 바이트코드를 추가한 방법 간의 프로그램 수행 시간이 어느 정도 개선되었는지 비교하였다. [그림 8]은 단일 루프를 사용해서 기존의 방법과 얼마 정도의 성능 차이가 있는지 측정하였다. 새로운 바이트코드의 추가를 통해서 기존의 방법보다 80-90% 정도의 성능 개선이 이루어진 것을 보여주고 있다.

(측정 단위: millisecond)



[그림 8] 단일 루프의 실행 시간 차이

[그림 9]는 중첩된 루프에서 기존의 방법과 새로운 바이트코드를 사용해서 수행한 시간의 차이를 보여주고 있다. 단일루프보다 더 많은 성능 개선이 이루어진 것을 볼 수 있으며 중첩된 루프의 수가 많아 질수록 더 많은 성능 개선이 이루어진 것을 볼 수 있다. 새로운 바이트코드의 추가를 통해서 이중 중첩된 루프나 삼중 중첩된 루프는 기존의 성능보다 약 80 – 90 %의 성능이 개선되는 것을 볼 수 있었다.

(측정 단위: millisecond)

루프의횟수	이중 중첩 루프	삼중 중첩 루프
100	0	164
200	7	1344
300	14	4217
400	27	10682
500	42	21847
600	60	35986
700	82	57120
800	106	85218
900	135	121283
1000	166	166337

[그림 9] 중첩된 루프에 따른 실행 시간 차이

5. 결론 및 연구 방향

본 논문에서는 자바프로그램에서 루프를 수행할 때 루프의 실행 횟수만큼 반복적으로 수행되는 바이트코드들의 비효율성을 개선하는 방법을 제시하였다. 제시한 방법은 자바가상머신에서는 아직 정의되지 않은 바이트코드가 있기 때문에 새로 정의된 루프 바이트코드를 제안하였고 이를 적용 해서 실제적인 구현을 통한 성능 평가를 하였다. 성능 평가를 통하여 기존의 방식보다 성능이 개선된 것을 볼 수 있었다.

또한 새로운 바이트코드를 추가함으로서 기계어 코드로 변환하는 플랫폼에 의존적인 복잡한 방법 대신 후처리기를 사용해 자바가상머신 내부에서 처리해주는 비교적 간단한 방법으로 자바가상머신의 성능을 개선할 수 있었다.

향후 연구 과제로는 본 논문에서 제시한 새로운 바이트코드인 Floop 이외에 비효율적인 방법으로 수행되는 다른 바이트코드들을 발견해서 자바가상머신의 성능 개선을 할 수 있는 바이트코드들을 정의하는 연구가 필요하다.

참고문헌

[1] Tim Lindholm and Frank Yellin, The Java Virtual Machine, Addison Wesley, 1997

[2] Bill Venners, Inside the Java Virtual Machine, McGraw-Hill, 1999

[3] James Gosling, The Java Language Specification, Addison-Wesley, 1996

[4] Joshua Engel, Programming for the Java Virtual Machine, Addison-Wesley, 1999

[5] 황순명, 조창오, 오세만, 자바 바이트코드 최적화기의 설계, 한국정보처리학회 98 추계 학술발표 논문, 5 권 2 호, pp 1264~1267, 1998