韓國 OA學會 論文誌
第6卷 第4號, 2001. 12.

2001-6-4-3-5

JIT Code Generator 상의 스택할당 정책 적용에 관한 연구

김 효 남*

A study of the stack allocation policy on JIT Code Generator

Hyo-Nam Kim*

요 약

자바프로그램의 실행속도를 빠르게 하는데 있어서 가장 좋은 방법은 빠른 자바가상머신(JVM : Java Virtual Machine)을 사용하는 것이다. 자바가상머신의 성능은 구현 차이에 따라 성능차이가 많이 난다. 자바가상머신을 구현하는데 있어서 가장 중요한 성능 향상의 기술은 JIT(Just-in-Time) 코드 생성기(Code Generator)이다. JIT 코드 생성기는 자바 바이트 코드를 플랫폼에 맞는 native machine code로 변환해 준다. 이 native code들은 자바가상머신에서 각 바이트 코드를 분석하는데 걸리는 시간을 단축할 수 있기 때문에 기존의 방식보다 빠르게 동작한다. 그러나 JIT 코드 생성기는 많은 레지스터를 사용하기 때문에 스택과 레지스터간의 traffic이 가중되는 문제가 있다. 그러므로 본 논문에서는 자바가상머신의 성능 향상을 위한 방안으로 효율적인 stack allocation 정책을 JIT 코드 생성기에 적용하여 레지스터와의 traffic을 감소시킬 수 있는 방안을 제시하였다.

Abstract

The best solution to improve the execution speed of Java program is to make use of the high speed JVM(Java Virtual Machine). The performance of JVM depends on the difference of its implementation. One of the technologies to enhance JVM performance is a JIT(Just-in-Time) code generator. The JIT code generator transforms Java byte code to the native machine code in accordance with computer system platform. The native machine code is faster than the existing interpreter method, since it can reduce the time to analyze the Java byte code. But the JIT code generator have the problem of increasing the traffic between stack and register because of using many register.

Therefore, this paper suggests how to reduce the traffic by applying the policy of stack allocation to JIT code generation, as one of the methods to enhance the performance of JVM.

* 청강문화산업대학 컴퓨터소프트웨어과

Ⅰ. 서론

Java Virtual Machine(JVM)은 Java 2 Platform을 구성하는 요소 중의 하나이다[1] JVM은 Class 파일 형태의 자바 바이트 코드를 실행하기 위한 환경이 된다. 그러므로 이런 JVM이 있는 어느 디바이스에서든지 Class 파일 형태의 자바 바이트 코드를 번역하여 실행할 수 있게 된다. 이런 이유로 자바가 이식성이 강한, 플랫폼에 독립적인 언어라는 특성을 가질 수 있는 것이다.

Class 파일 형태의 자바 바이트 코드를 실행하기 위해서는 JVM상에서 기계가 인식할 수 있는 machine code로 바꿔주는 인터프리터의 과정을 거치게 된다.

그러나 기존의 인터프리터는 자바 바이트 코드를 차례로 읽어 machine code로 번역하여 수행하게 되므로 machined code로의 번역시간이 많이 소요되는 단점을 갖고있다. 자바 기술과 관련하여 실행속도 향상을 위하여 JVM상의 인터프리터 시간을 줄이고자 고안된 것이 JIT (Just-in-Time) 코드 생성기(Code Generator)이다.

대부분의 언어에 있어서 컴파일러는 컴파일 시에 소스 코드로부터 native machine code를 생성해 내지만 이러한 코드들은 해당 플랫폼(Platform)에 종속적일 수밖에 없다. JIT 코드 생성기는 클래스가 로드된 후에 프로그램을 native machine code로 변환한다. 즉, native machine code는 실제로 필요하기 전까지는 생성되지 않는다. 이는 프로그램이 실행될 때의 컴퓨터 환경에 맞춰서 최적화될 수 있다는 것을 의미한다. JIT 코드 생성기는 컴파일 시에 각 명령어에 대해서 많은 최적화를 적용할 수 있다는 장점을 가질 수 있다. JIT에서는 심지어 프로그램이 수행 중인 중간에도 최적화를 할 수 있다.

그리고 JIT 코드 생성기는 스택을 기반으로 하는 자바 바이트 코드를 레지스터 기반의 native machine code로 변환 시켜주는 코드 생성기이다[3]. JIT 코드 생성기가 자바 바이트 코드를 기계가 바로 인식할 수 있는 레지스터 기반의 native machine code를 생성시켜 Java virtual machine상의 번역시간을 줄이므로 해서 자바 프로그램 실행 속도를 향상시켜준다.

그러나 JIT code generator는 적어도 7개 이상의 많은 레지스터가 사용된다[4]. 그러므로 많은 레지스터 사이에서의 효율적인 allocation 정책이 필요하게 된다. 또한, 스택기반의 자바 바이트 코드를 JIT 코드 생성기를 통하여 레지스터 기반의 native machine code로 변환하기 위해서 스택과 레지스터 사이의 많은 로딩이 있게 되므로 그에 따르는 load traffic이 증가하게 된다.

본 논문에서는 스택을 기반으로 하는 자바 바이트 코드를 스택에 저장하는데 효율적인 스택 allocation정책을 JIT 코드 생성기에 적용함으로 자바 바이트 코드를 레지스터 기반의 native machine code로 생성할 때 스택과 레지스터의 간의 load 줄이므로 load traffic을 감소시키는 방법을 제시하고 있다.

Ⅱ. 본 론

위에서 언급한 바와 같이 자바 바이트 코드는 스택을 기반으로 하고 있다. 자바 바이트 코드를 레지스터를 기반으로 하는 native machine code로 변환하는 JIT code generator는 5가지의 pass로 진행이 되어진다.[4]

1) Prepass : 이 과정에서 바이트 코드에 관한 정보를 수집한다.
2) Global register allocation : 이 부분에서는 local variable과 operand들을 register에 allocation하는 과정을 수행한다.
3) Code generation : 부분에서 자바 바이트 코드를 native machine code로 변환 한다.
4) Code emission : code generation의 결과를 메모리에 저장한다.
5) Code and data patching : 실제적으로 native machine code가 실행되어진다.

그림 1은 JIT code generator의 다섯가지 pass 동작을 그림으로 표현한 형태이다.

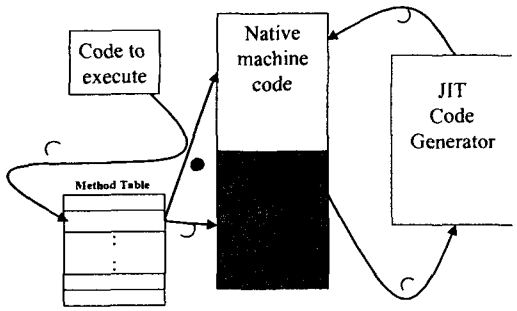

그림 1. JIT code generator의 동작

기존의 JIT code generator는 두 번째 pass, register allocation하는 과정에서 많은 스택과 레지스터 사이의 로딩이 생기게 되고 load traffic이 발생하게 된다. 스택과 레지스터간의 load traffic을 최소화하기 위해서 효율적인 allocation 정책이 필요하다.

앞에서 말한 바와 같이 이 논문에서는 두 번째 pass에서 발생하는 스택과 레지스터 traffic을 최소화하기 위해서 첫번째 단계인 Prepass과정, 즉 자바 바이트 코드에 관한 정보를 수집하는 과정에 효율적인 stack allocation을 적용함으로 자바 바이트 코드를 효율적으로 location하도록 한다. 이런 첫번째 단계, 자바 바이트 코드를 스택에 효율적으로 allocation을 수행한 후의 자바 바이트 코드는 Global allocation단계에서 local variable 레지스터와 operand를 레지스터로 allocation 하는 과정에 발생하는 load traffic을 줄이는 효과를 가져오게 된다.

효율적인 stack allocation을 하기 위해서 이중으로 load 되어야 하거나 스택에 load된 constant value를 다음 연산을 위해 저장되어야 하는 문제를 해결하므로 load를 줄이는 방법이다. 이를 위해서는 dup 명령을 사용한다.

그림 2는 자바 소스 코드와 stack allocation 정책을 적용 하기전의 자바 바이트 코드와 stack allocation을 적용하고 난 후의 자바 바이트 코드이다.

Source code	Before stack allocation	After stack allocation
b = a * a;	iload Local$1 iload Local$1 imul istore Local$2	iload Local$1 dup imul istore Local$2

그림 2. Stack allocation 정책 적용전과 후의 자바 바이트 코드

자바 바이트 코드에 효율적인 Stack allocation을 적용하기 전에는 Local$1을 두 번 load하고 있다. 이 자바 바이트 코드가 JIT code generator를 통해서 native machine code로 바뀌면 local value resister에서 local value를 읽어 오는데 이 과정을 두 번 수행하므로 레지스터 access를 두 번 하게 된다. 이렇게 같은 local value를 access하기 위한 반복적인 load를 줄이기 위해서 dup을 사용하여 효율적인 stack allocation을 적용한다. 오른쪽의 자바 바이트 코드가 효율적인 stack allocation을 적용한 자바 바이트 코드의 스택이다. 이 자바 바이트 코드를 JIT code generator를 통하여 native machine code로 변환을 하면 local value 레지스터를 한번 access하여 local value를 load하고 동일한 local value를 load하기 위해 dup명령을 사용하여 전에 load한 local value를 복사한다. 원래의 local value register에 있는 value와 그를 dup명령으로 복사한 동일한 value로 다음 명령인 imul을 수행하게 된다.

이렇게 그림 2는 자바 바이트 코드에 대한 stack allocation을 적용하지 않았을 경우에는 두 번의 load와 한 번의 store를 위해서 레지스터를 세 번access해야 하는 결과를 보여준다. 그러나 반복적으로 동일한 local value를 load하는 경우는 dup라는 명령을 사용하여 반복적인load 과정을 제거하는 효율적인 stack allocation을 적용함으로 한번의 local value의 load와 결과 값 저장을 위한 store로 크게 두 번 레지스터를 access하게 된다. 결과적으로 스택과 레지스터간의 load traffic을 줄이게 된다.

Source code	Before stack allocation	After stack allocation
b = (a + 5) / 5;	iload Local$1 ldc 5 iadd iload Local$1 idiv istore Local$4	iload Local$1 dup ldc 5 iadd swap idiv istore Local$4

그림 3. 다른 load로부터 연산 된 결과를 유지하기 위해 적용된 Stack allocation 정책

또한, 이미 load 되어 연산 된 값을 다른 value들의 load로부터 유지하기 위해 임의의 스택 공간에 저장하는 swap명령을 사용한다.

그림 3은 다른 local value의 load로부터 그 value를 유지하기 위해서 swap명령을 사용한 예이다. dup명령으로 local value a의 반복 load의 수를 줄이고 우선

iadd명령을 수행한다. 다음 명령인 idiv을 수행하기 전에 이미 load된 local value a로부터 iadd한 결과를 유지하기 위해서 swap명령을 수행한다. 이렇게 그림 2에서 보여주는 바와 같이 JIT code generator에서 자바 바이트 코드에 효율적인 stack allocation을 적용하기 전에는 레지스터를 세 번 access하지만 효율적인 stack allocation을 적용한 후에는 두 번의 access로 연산이 수행되므로 레지스터를 access하기 위한 스택의 load명령을 줄이므로 스택과 레지스터 사이의 load traffic을 줄이는 효과를 가져온다.

Ⅲ. 결 론

본 논문은 JIT code generator의 수행단계의 첫번 단계인 Prepass에 효율적인 stack allocation을 적용함으로 스택 기반인 자바 바이트가 JIT code generator를 통해서 레지스터 기반의 native machine code로 변환하는 과정에 생길 수 있는 스택과 레지스터간의 load traffic을 줄이고자 하였다.

그러나 레지스터와 스택의 처리 속도와 비용을 고려하여 어느 정도의 레지스터의 사용과 스택의 사용이 JIT code generator를 통한 인터프리터의 번역시간을 최상으로 줄일 수 있는지에 관한 연구가 더불어 함께 실행되어야 한다.

참고문헌

[1] Tim. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, September 1996.

[2] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[3] Andreas Krall. Efficient Java VM Just-in-time Compilation. In proceedings of PACT.98. 1998. Http://www.complang.tuwien.ac.at/andi/

[4] Bill Venners, "Inside the Java Virtual Machine", McGraw-Hill, 1998

[5] John Meyer, Troy Dowing, "Java Virtual Machine", O' RELLY, 1997

[6] Ali-Reza dl-Tabatabai et. al. Fast, Efficitive Code Generation in a Just-in-Time Java Compiler. In proceedings of ACM PLDI ,98. Jun 1998.

[7] Byung-Sun Yang, Soo-Mook Moon et. Latte : A java VM Just-in Time Compiler with Fast and Efficient Register Allocation. Seoul National University.1999

[8] Tim. Lindholm and F. Yellin. The Java Virtual Machine Specification Second Edition. Addison-Wesley, 2000.

저 자 소 개

김 효 남

1988년 홍익대학교 전자계산학과 (이공학사)

1990년 홍익대학교 전자계산학과 (이공석사)

1999년 홍익대학교 전자계산학과 (박사과정)

현재 청강문화산업대학 컴퓨터소프트웨어과 교수

관심분야 : Programming Language, Object Oriented Programming, XML/EDI, XSLT