2001 년 한국정보처리학회 추계 학술발표논문집 제 8 권 제 2 호

레지스터 로드 트래픽 감소를 위한 JIT Code Generator 에 스택할당 정책 적용 방안 연구

°송경남* 김효남** 원유헌*
홍익대학교 컴퓨터공학과* 청강문화산업대학 소프트웨어학과**
{knsong, won}@cs.hongik.ac.kr*, hnkim@mail.chungkang.ac.kr**

A study stack allocation on JIT Code Generator for reducing register load traffic

°Kyung-nam Song* Hyo-nam Kim** Yoo-hun Won*
Department of Computer Engineering, Hong-Ik University*.
Department of Computer Software, Chung Kang College of Cultural Industries**

요 약

Java virtual machine 의 성능을 향상 시키기 위해 "JIT(Just-in-Time)"code generator 가 고안되었다[3]. JIT code generator 는 스택기반의 자바 바이트 코드를 레지스터 기반의 native machine code 로 변환해 주는 역할을 수행하여 바이트 코드의 번역시간을 줄여준다. 그러나 JIT 는 많은 레지스터의 사용을 야기시키므로 효율적인 레지스터 allocation 정책이 필요하고 스택과 레지스터 간의 traffic 을 가중시킨다. 그러므로 본 논문에서는 자바 바이트 코드의 효율적인 stack allocation 정책을 JIT code generator 에 적용함으로 레지스터와의 traffic 을 줄이는 방법을 제시하였다.

1. 서론

Java virtual machine(JVM)은 Java 2 Platform 의 을 구성하는 요소중의 하나이다[1]. JVM 은 Class 파일 형태의 자바 바이트 코드를 실행하기 위한 환경이 된다. 그러므로 이런 JVM 이 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 JVM 상에서 기계가 인식할 수 있는 machine code 로 바꿔주는 인터프리터의 과정을 거치게 된다.

그러나 기존의 인터프리터는 자바 바이트 코드를 차례로 읽어 machine code 로 번역하여 수행하게 되므로 machined code 로 번역 시간이 많이 소요되게 된다. 이렇게 JVM 상의 인터프리터 시간을 줄이고자 고안된 것이 "JIT(Just-in-Time)"code generator 이다.

JIT code generator 는 스택을 기반으로 하는 자바 바이트 코드를 레지스터 기반의 native machine code 로 변환 시켜주는 코드 생성기이다[3].

자바 바이트 코드를 JIT code generator 를 통해 레지스터 기반으로 기계가 바로 인식할 수 있는

native machine code 로 변환을 시켜주어 Java virtual machine 상의 번역 시간을 향상시켜준다.

그러나 JIT code generator 는 적어도 7개 이상의 많은 레지스터가 사용된다[4]. 그러므로 많은 레지스터 사이에서의 효율적인 allocation 정책이 필요하게 된다. 또한, 스택기반의 자바 바이트 코드를 JIT code generator 를 통하여 레지스터 기반의 native machine code 로 변환하기 위해서 스택과 레지스터 사이의 많은 로딩이 있게 되므로 그에 따르는 load traffic 이 증가하게 된다.

본 논문에서는 스택을 기반으로 하는 자바 바이트 코드를 스택에 location 하는데 효율적인 스택 allocation 정책을 JIT code generator 에 적용함으로 자바 바이트 코드를 레지스터 기반의 native machine code 로 생성할 때 스택과 레지스터의 간의 load 줄이므로 load traffic 을 감소시키는 방법을 제시 하고 있다.

2.본 론

위에서 언급한 바와 같이 자바 바이트 코드는 스택을 기반으로 하고 있다. 자바 바이트 코드를 레지스터를 기반으로 하는 native machine code 로 변환하는 JIT code generator 는 5 가지의 pass 로 진행이 되어진다[4].

1) Prepass : 이 과정에서 바이트 코드에 관한 정보를 수집한다.
2) Global register allocation : 이 부분에서는 local variable 과 operand 들을 register 에 allocation 하는 과정을 수행한다.
3) Code generation : 부분에서 자바 바이트 코드를 native machine code 로 변환 한다.
4) Code emission : code generation 의 결과를 메모리에 저장한다.
5) Code and data patching : 실제적으로 native machine code 가 실행되어진다.

기존의 JIT code generator 는 두 번째 pass, register allocation 하는 과정에서 많은 스택과 레지스터 사이의 로딩이 생기게 되고 load traffic 이 발생하게 된다. 스택과 레지스터 간의 load traffic 을 최소화하기 위해서 효율적인 allocation 정책이 필요하다.

앞에서 말한 바와 같이 이 논문에서는 두 번째 pass 에서 발생하는 스택과 레지스터 traffic 을 최소화하기 위해서 첫번째 단계인 Prepass 과정, 즉 자바 바이트 코드에 관한 정보를 수집하는 과정에 효율적인 stack allocation 을 적용함으로 자바 바이트 코드를 효율적으로 location 하도록 한다. 이런 첫번째 단계, 자바 바이트 코드를 스택에 효율적으로 allocation 을 수행한 후의 자바 바이트 코드는 Global allocation 단계에서 local variable 레지스터와 operand 를 레지스터로 allocation 하는 과정에 발생하는 load traffic 을 줄이는 효과를 가져오게 된다.

효율적인 stack allocation 을 하기 위해서 이중으로 load 되어야 하거나 스택에 load 된 constant value 를 다음 연산을 위해 저장되어야 하는 문제를 해결하므로 load 를 줄이는 방법이다. 이를 위해서는 dup 명령을 사용한다.

그림 1 은 자바 소스 코드와 stack allocation 정책을 적용 하기전의 자바 바이트 코드와 stack allocation 을 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

그 림 1

이중으로 load 되어야 하는 value 에 적용된 효율적인 allocation 정책

자바 바이트 코드에 효율적인 Stack allocation 을 적용하기 전에는 Local$1 을 두 번 load 하고 있다. 이 자바 바이트 코드가 JIT code generator 를 통해서 native machine code 로 바뀌면 local value resister 에서 local value 를 읽어 오는데 이 과정을 두 번 수행하므로 레지스터 access 를 두 번 하게 된다. 이렇게 같은 local value 를 access 하기 위한 반복적인 load 를 줄이기 위해서 dup 을 사용하여 효율적인 stack allocation 을 적용한다. 오른쪽의 자바 바이트 코드가 효율적인 stack allocation 을 적용한 자바 바이트 코드의 스택이다. 이 자바 바이트 코드를 JIT code generator 를 통하여 native machine code 로 변환을 하면 local value 레지스터를 한번 access 하여 local value 를 load 하고 동일한 local value 를 load 하기 위해 dup 명령을 사용하여 전에 load 한 local value 를 복사한다. 원래의 local value register 에 있는 value 와 그를 dup 명령으로 복사한 동일한 value 로 다음 명령인 imul 을 수행하게 된다.

이렇게 그림 1 은 자바 바이트 코드에 대한 stack allocation 을 적용하지 않았을 경우에는 두 번의 load 와 한 번의 store 를 위해서 레지스터를 세 번 access 해야 하는 결과를 보여준다. 그러나 반복적으로 동일한 local value 를 load 하는 경우는 dup 라는 명령을 사용하여 반복적인 load 과정을 제거 하는 효율적인 stack allocation 을 적용함으로 한번의 local value 의 load 와 결과값 저장을 위한 store 로 크게 두 번 레지스터를 access 하게 된다. 결과적으로 스택과 레지스터간의 load traffic 을 줄이게 된다.

Source code	Before stack allocation	After stack allocation
b = (a + 5) / a;	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

그 림 2

다른 load 로부터 연산 된 결과를 유지하기 위해 적용된 allocation 정책

또한, 이미 load 되어 연산 된 값을 다른 value 들의 load 로부터 유지하기 위해 임의의 스택 공간에 저장하는 swap 명령을 사용한다.

그림 2 는 다른 local value 의 load 로부터 그 value 를 유지하기 위해서 swap 명령을 사용한 예이다. dup 명령으로 local value a 의 반복 load 의 수를 줄이고 우선 iadd 명령을 수행한다. 다음 명령인 idiv 을 수행하기 전에 이미 load 된 local value a 로부터 iadd 한 결과를 유지하기 위해서 swap 명령을 수행한다. 이렇게 그림 2 에서 보여주는 바와 같이 JIT code generator 에서 자바 바이트 코드에 효율적인 stack allocation 을 적용하기 전에는 레지스터를 세 번 access 하지만 효율적인 stack allocation 을 적용한 후에는 두 번의 access 로 연산이 수행되므로 레지스터를 access 하기 위한 스택의 load 명령을 줄이므로 스택과 레지스터 사이의 load traffic 을 줄이는 효과를 가져온다.

3. 결 론

본 논문은 JIT code generator 의 수행단계의 첫 번 단계인 Prepass 에 효율적인 stack allocation 을 적용함으로 스택 기반인 자바 바이트가 JIT code generator 를 통해서 레지스터 기반의 native machine code 로 변환하는 과정에 생길 수 있는 스택과 레지스터 간의 load traffic 을 줄이고자 하였다.

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

4. 참 고 문 헌

[1] T. 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]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.

[5] 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*