스택 기반 코드 변환기 설계

김경수, 김기태, 조선문, 심현진, 유원희
인하대학교 컴퓨터 정보공학과
e-mail: oe0916@hanmail.net

Design of Translator for Stack-Based Code

Kyung-Soo Kim, Ki-Tae Kim, Sun-Moon Jo,
Hyun-Jin Sim, Weon-Hee Yoo
Dept. of Computer Science and Engineering, Inha University

요 약

자바는 객체 지향언어로 네트워크 기반의 환경에서 응용프로그램을 효율적으로 개발을 위해 설계되었으며 특정 하드웨어나 운영체제에 영향을 받지 않고 동작 할 수 있는 높은 이식성을 가지고 있다. 하지만 자바 언어로 응용프로그램을 개발하면 다른 언어로 작성하는 것 보다 일반적으로 실행이 느리다는 단점이 발생하게 된다. 이를 극복하기 위해 자바 언어에 대한 최적화가 필요하다. 일반적으로 많이 쓰는 최적화 방법으로는 JIT와 같이 실행시간에 필요에 따라 컴파일하여 실행하는 방법과 바이트코드를 수행하고자 하는 특정 프로세서에 적합한 네이티브 코드를 생성하는 방법을 사용하고 있다. 하지만 이러한 방법들은 고유의 최적화 기법과 구현된 프로그램을 이용해서만이 최적화가 가능했고 또한 코드의 추출부터 최적화를 위한 모든 부분까지 구현해야하는 문제점이 있다. 또한 자바 바이트코드는 스택기반의 코드로써 명령어가 많고 표현이 명백하지 않다는 문제점을 가지고 있다. 따라서 분석과 변환이 어렵기 때문에 자바 바이트코드를 스택 기반이 아닌 분석과 최적화가 용이한 3-주소 형태로 변환하여 최적화 기법을 적용해야 한다 본 논문에서는 자바 바이트코드와 3-주소 형태의 중간 코드인 CTOC-B에 대해서 설계한다.

1. 서론

자바는 객체 지향언어로 네트워크 기반의 환경에서 응용 프로그램을 효율적으로 개발을 위해 설계되었으며 특정 하드웨어나 운영체제에 영향을 받지 않고 동작 할 수 있는 높은 이식성을 가지고 있다[1]. 하지만 자바로 응용 프로그램을 개발하면 다른 언어로 작성하는 것 보다 일반적으로 실행이 느리다는 단점을 가진다. 그 이유는 자바는 프로그래밍 환경에서 이기종간의 실행환경에 적합하도록 자바 가상 기계에서 바이트코드가 인터프리터 방식으로 사용되기 때문이다[2]. 따라서 작은 크기의 자바 응용 프로그램 수행에는 실행속도 문제가 중요한 요소가 아니지만 대형프로그램의 수행에는 실행 속도가 현저히 저하되는 단점이 발생하게 된다. 현재 실행속도의 개선을 위해 많은 방법들이 개발되고 있다. 그 중 JIT 방식은 실행시간에 필요에 따라 컴파일하여 실행하는 방법과 전통적인 컴파일 방식을 사용하여 바이트코드를 수행하고자 하는 특정 프로세서에 적합한 네이티브 코드를 생성하는 방법을 사용하고 있다[3][4]. 그 외에도 여러 가지 연구들과 방법들이 제시 되고 있지만 이러한 방법들은 구현된 프로그램을 이용해서만이 최적화가 가능하고 또한 코드의 추출

부터 최적화를 위한 모든 부분을 구현해야하는 문제점을 가진다.

자바 바이트코드는 스택 기반의 코드로써 200여개나 되는 명령어를 가지고 있고, 많은 명령어로 인해 명확한 표현이 어렵다는 문제점을 가지고 있어 최적화시 분석과 변환이 어렵기 때문에 자바 바이트코드를 스택 기반이 아닌 분석과 최적화가 적합한 3-주소 형태로 변환하여 최적화 기법을 적용하면 더욱 효율적인 최적화된 코드를 생성할 수 있게 된다.

본 논문의 구성은 2장에서는 스택 기반 바이트코드에 대해 소개를 하고 3장에서는 스택 기반 바이트코드를 3-주소 형태로 변환하기 위한 중간코드인 CTOC-B(Class To Optimized Class-Bytecode)의 설계에 대해 목적을 두고 있다. 4장에서는 결론 및 향후 연구 과제에 대하여 논의할 것이다.

2. 관련 연구

자바 가상 기계는 스택 기반의 기계이다. 따라서 바이트코드에는 스택에 대한 동작들이 정의되어 있고 현재 바이트코드는 200개가 넘는 명령어로 구성되어 있다. 기존의 자바 바이트코드는 스택 기반의 중간 표현을 이용해서 자바 가상머신에서 동작한다[5]. 스택 기반의 바이트코드를 중간표현으로 사용했을 때 두 가지의 장점이 있는데 그 중 첫 번째 장점은 스택 코드로 생산된 결과들은 변환이 필요가 없이 사용 할 수 있다는 점이다. 즉 이전 동작에 의해 결과로 생성된 스택 코드는 추가적인 어떠한 변화 없이도 스택 머신에서 수행이 가능하게 때문에 생성된 코드에 대한 변환이 필요없다는 것이다. 두 번째 장점은 스택 코드를 즉시 사용할 수 있다. 자바 클래스 파일은 스택 기반 코드이기에 스택 코드에 대한 변환없이 사용이 가능하다.

하지만 이러한 스택 기반 코드를 분석이나 최적화를 위해 사용했을 때 몇 가지 단점이 발생할 수 있다. 그 단점으로는 첫 번째, 표현이 명확하지 않다는

iconst_3 iload_2 iadd istore_0	a=3+b
바이트코드	3-주소 코드

[그림2-1] 바이트코드에서 3-주소 코드의 변환

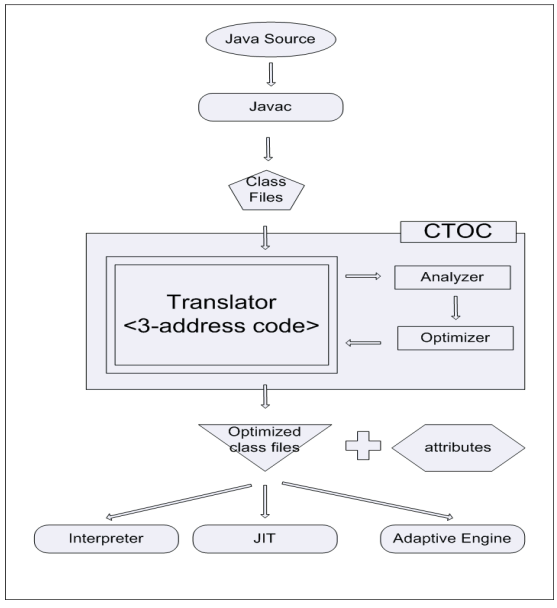
것이다. 명확한 x=a+b와 같은 3-주소 형태와는 다르게 스택 기반의 자바 바이트코드는 오퍼랜드 스택에 영향을 주는 iload, iadd, imul, pop과 같은 명령어와 필드를 수정하는 putfield, 메서드를 호출하는 invokestatic, 지역변수에 저장하는 istore와 같은 명령어로 표현되기에 표현들이 명확하지 않게 된다.

두 번째, 스택 기반 코드는 단순한 변환을 복잡하게 만들 수 가 있다. 스택 기반 코드가 분석과 변형을 복잡하게 만드는 주된 이유는 바이트코드의 단편적인 형식 때문이다. 이런 형태에서는 모든 분리된 코드를 유지해야 할 필요가 있기 때문에 코드를 변환하는 것이 매우 어렵다. [그림 2-1]에서와 같이 바이트코드의 변환과 분석을 하기 위해서는 기존의 스택 코드를 사용하는 것 보다 3-주소 형태의 코드를 사용하는 것이 더욱 효율적이다.

세 번째, 동일한 표현에 많은 명령어를 표현하게 된다. 예를 들어 var5 * var4 / (var3 + var7) 과 같은 연산을 할 때 스택 기반 코드의 경우 8개의 명령어를 이용해서 처리해야 하기 때문에 표현에 많은 명령어를 사용해야 한다. 3-주소 형태의 코드로 변화 시키면 효율적으로 처리 할 수 있게 된다.

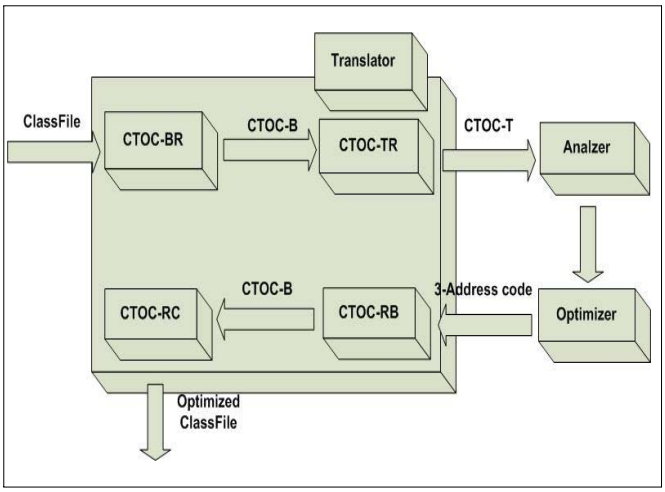
마지막으로 스택 기반 기계는 값을 재사용 할 수 없기 때문에 불필요한 적재(load)와 저장(store) 명령어가 많아지게 되어 분석과 판독이 어렵게 되는 단점을 발생하게 된다. 이러한 스택 기반의 중간 표현은 3-주소 형태의 중간 표현으로 변환하면 분석이나 최적화에 효율적으로 적용할 수 있게 된다.

3. CTOC-B의 설계



[그림 3-1] CTOC의 구성도

스택 기반의 바이트코드를 3-주소 형태인 CTOC-T(Class To Optimized Class-Three address code)를 만들기 위한 중간 표현 형태인 CTOC-B를 설계하려고 한다. [그림 3-1]은 CTOC의 전체적인 구성을 보여주고 있다.



[그림 3-2] CTOC의 내부 구성도

CTOC-B는 스택 기반의 바이트코드 표현으로써 분석과 변환을 좀더 단순화하는 것이다.

우선 CTOC의 전체적인 흐름을 살펴보면 [그림 3-2]에 나온 것과 같이 자바 클래스 파일을 CTOC-BR(Class To Optimized Class-Bytecode tRanslator) 변환기에 의해 CTOC-B가 생성 된다.

CTOC-B는 아직 스택 기반의 코드이며, 스택 기반 코드와 3-주소 형태의 중간 표현이다. 이렇게 생성

[표 3-1] CTOC-B의 명령어 리스트

local:=@zero local:=@exception T.dup1 T.add T.and T.cmpg T.or T.load *local* T.store *local*	goto *local* return *local* T.pop breakpoint nop push *constant* lookupswitch fldget *field* ifeg *label*

된 CTOC-B는 CTOC-TR이라는 3-주소 형태로 변환하는 변환기 의해 CTOC-T로 만들어 진다. CTOC-T는 분석과 변환이 용이하기 때문에 최적화를 위해서 사용된다. 최적화된 CTOC-T는 CTOC-RC(Class To Optimizer Classes -Return ClassFile)을 거쳐 CTOC-B가 되고 CTOC-B는 CTOC-RC(Class To Optimizer Classes Return ClassFile)를 거쳐 클래스파일로 변환된다.

CTOC-BR은 바이트코드를 CTOC-B로 패턴 매칭 기법으로 변환한다. CTOC-B는 스택 기반의 바이트코드의 중간표현으로 [표 3-1]에서 보듯이 CTOC-B는 스택 기반의 바이트코드의 200여개의 명령어들을 타입화 함으로써 그 수를 줄였다. 예를 들면 I.add, D.add, R.add와 같이 명령어 앞에 타입을 대문자로 표기함으로써 타입화를 하였다.

CTOC-B 변환을 하면서 형태적 변화들에 대하여 살펴보면 우선 바이트코드에서 상수 및 필드, 메서드, 클래스의 접근을 하기 위해 사용했던 상수 풀을 CTOC-B에서는 내부적으로 직접적으로 표현함으로써 코드 조작을 쉽게 만들었다. 예를 들면

bytecode : ldc #4 // string five

CTOC-B : push "five"

와 같이 상수 풀에 있는 내용을 "five"와 같이 직접

[표 3-2] 자바 바이트코드에서 CTOC-B 코드변환

	var args
0: iconst_2	push 2 ;
1: newarray int	newarray ;
3: astore_1	R.store args ;
4: aload_1	R.load args ;
5: iconst_0	push 0 ;
6: iconst_1	push 1 ;
7: iastore	I.arraywrite ;
8: aload_1	R.load args ;
9: iconst_1	push 1 ;
10: iconst_0	push 0 ;
11: iastore	I.arraywrite ;
12: aload_1	R.load args ;
13: iconst_0	push 0 ;
14: iaload	I.arrayread ;
15: aload_1	R.load args ;
16: iconst_1	push 1 ;
17: iaload	I.arrayread ;
18: iadd	I.add ;
19: istore_2	I.store args;
20: return	return ;
자바 바이트코드	CTOC-B 코드

적으로 표현을 한다.

두 번째 특징은 goto *local*을 사용하여 복잡해질 수 있는 분석과 변형을 많은 부분에서 단순화 될 수 있게 만들었다.

세 번째 특징은 CTOC-B에서는 바이트코드의 지역 변수를 명시적으로 표현하게 된다. 즉 지역 변수에 대해서 32-bit를 정의하는 var 타입을 사용하고 있다. CTOC-B에서의 변환 이후 3-주소 형태의 변환된 CTOC-T 에서는 더욱 명확한 지역 변수의 이름을 허용하게 된다.

[표 3-2]에서는 바이트코드에서 CTOC-BR을 통해서 CTOC-B로의 변환된 코드로의 변환을 보여준다. CTOC-B는 단지 3-주소 형태로 변환을 쉽게 하기 위한 변환만을 하고 특별한 최적화 기법이 들어가지는 않는다. 이후 CTOC-B는 CTOC-TR을 거친 후 CTOC-T로 3-주소 형태로 변환이 다시 이루어지고 CTOC-T에서 3-주소 형태에서 최적화 기법들이 적용되어 최적화를 이루게 된다.

4. 결론 및 향후 연구

자바의 실행 속도를 빠르게 하기 위한 방법들은 많이들 제시되었다. 본 논문에서는 실행속도를 빠르게 하기위해 스택 기반의 바이트코드를 직접적으로 변환하여 분석 간단한 CTOC-B를 설계하였다. CTOC-B는 바이트코드와 유사한 스택 기반의 코드이며 단지 3-주소로 가기 위한 중간 표현이다. CTOC-B에서는 변환만을 했을 뿐 실질적인 최적화 기법과 분석은 CTOC-TR에 의해 만들어진 CTOC-T에서 이루어지게 된다. CTOC-T는 3-주소 코드의 형태로서 분석이 편하고 최적화 기법을 이용하고 적용을 할 수가 있다.

앞으로 CTOC의 연구 과제는 최적화된 바이트코드를 프레임 워크에서 사용 할 수 있게 연구할 예정이다.

참고문헌

[1] Ken Arnold and James Gosling, *"The Java Programming Language"*, Sun Microsystem, 1996.

[2] Tim Lindholm and Frank Tellin. *"The Java Virtual Machine Specification."* Morgan Kaufmann, 1997.

[3] John Meyer, Troy Downing, *"Java Virtual Machine"*, O'RELLAY, 1997.

[4] Wen-mei W. Hwu, *"Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results"*, The proceeding of the 29th Annual International Symposium on Microarchitecture, Dec, 1996.

[5] B.Venners, *"Inside the Java Virtual Machine"*, McGraw Hill, 1998.