제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

CTOC에서 스택 기반 코드를 효율적인 중간코드로 변환기 설계

김 경수, 김 기태, 조 선문, 유 원희
인하대학교 컴퓨터 정보공학과
e-mail:oe0916@hanmail.net

Design of Translator for Efficient Intermediated Code from Stack Based Codes in CTOC

Kyung-Soo Kim, Ki-Tae Kim, Sun-Moon Jo, Weon-Hee Yoo
Dept of Computer Science and Engineering, In-Ha University

요 약

자바 언어는 객체지향 언어이며 효율적인 애플리케이션을 개발하기 위해 설계되었다. 특히 다양한 개발 환경과 이식성에 맞는 언어로써 각광을 받고 있다. 하지만 자바 언어로 애플리케이션을 개발하면 다른 언어로 작성하는 것 보다 실행이 느리다는 단점을 가지고 있다. 이러한 자바 실행 속도를 극복하기 위해 많은 연구가 되고 있는데, 그 중에서도 JIT방식과 네이티브 코드로 변환 방식이 있다.

본 논문은 스택기반의 자바 바이트코드에서 3-주소 형태로 변환하여 최적화하는 CTOC중에서 바이트코드에서 3-주소 형태 즉 CTOC-T의 중간 표현인 CTOC-B를 설계하려 한다. CTOC-B는 스택기반의 중간표현으로써 자바 바이트코드보다 코드의 변환과 분석이 용이하게 만든 형태의 표현이다.

본 논문에서는 자바 바이트코드에서 스택기반 중간코드인 CTOC-B 코드로의 효율적인 변환기를 설계하며, CTOC-B의 특징을 분석해 본다.

1. 서론

자바 언어는 객체지향 언어이며 효율적인 애플리케이션을 개발하기 위해 설계되었다. 특히 플랫폼이 독립적이기에 다양한 개발 환경과 이식성에 맞는 언어로써 각광을 받고 있다[1]. 하지만 자바 언어로 애플리케이션을 개발하면 다른 언어로 작성하는 것 보다 일반적으로 실행이 느리다는 단점을 가지고 있다. 그 이유는 자바 프로그래밍 환경에서 자바 가상기계코드인 바이트코드가 인터프리터 방식으로 사용되기 때문이다[2,3]. 따라서 작은 크기의 자바 응용프로그램 수행에는 실행속도 문제가 중요한 요소가 아니지만 대형프로그램의 수행에는 실행 속도가 현저히 저하되는 단점이 발생하게 된다. 이러한 실행 속도의 문제를 극복하기 위해 많은 연구가 되고 있는데, 그중에서도 실행시간으로 메서드 단위를 컴파일 하는 JIT방식[4]과 네이티브 코드로 변환을 이용하는 JCC[5], CACAO[6] 등의 방식이 있다. 본 논문에서는 스택기반의 자바 바이트코드를 3-주소 형태로 변환하여 최적화하는 CTOC(Class To Optimized Class)중에서 CTOC-B(Class To Optimized Class-Bytecode)를 제안하려 한다. CTOC-B는 스택 기반의 자바 바이트코드가 3-주소 형태인 CTOC-T (Class To Optimized Class-Three address code)로 변환하기 위한 스택 기반의 중간 표현이다.

본 연구는 한국과학재단 목적기초연구(R05-2004-000-11694-0) 지원으로 수행되었음.

본 논문의 구성은 제 2장에서는 바이트코드와 CTOC, CTOC-B에 대한 소개를 하고 제 3장에서는 스택기반코드를 3-주소 형태로 변화하기위한 중간코드인 CTOC-B의 설계에 대해 목적을 두고 있다. 제 4장에서는 결론과 향후 과제에 대하여 논의할 것이다.

2. 관련연구

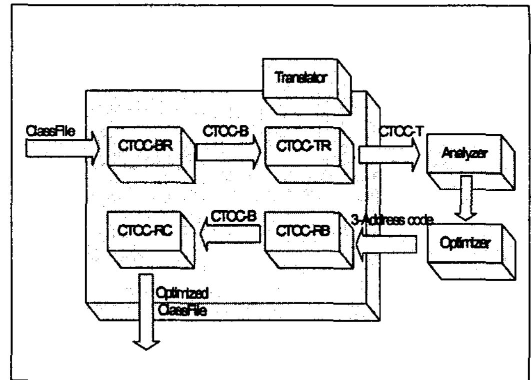
자바 바이트코드는 자바 개발 환경에서 컴파일 과정 이후 생성된 목적코드가 모든 플랫폼에 적용 될 수 있도록 코드를 생성하는데 이를 바이트코드라고 한다. 정확하게 말하면 컴파일 과정이후 클래스 파일이 생성되고 생성된 클래스 파일은 크게 12부분으로 나눌 수가 있는데 바이트코드는 클래스 파일 전체를 말하는 것은 아닌 클래스 파일에서도 메서드 어트리뷰트 부분에 배열형태로 표현되어 있는 부분을 바이트코드라고 한다.

자바 프로그램	바이트코드

public static int
stepPoly(int x){
if(x<0){
System.out.println(
"Bad Argument");
return -1;
}
else if(x <= 5){
return x * x;
}
else {
return x * 5 + 16;
}
}
	0: iload_0
	1: ifge14
	4: getstatic #25
	7: ldc #37
	9: invokevirtual #40
	12: iconst_m1
	13: ireturn
	14: iload_0
	15: iconst_5
	16: if_icmpgt 23
	19: iload_0
	20: iload_0
	21: imul
	22: ireturn
	23: iload_0
	24: iconst_5
	25: imul
	26: bipush 16
	28: iadd
	29: ireturn

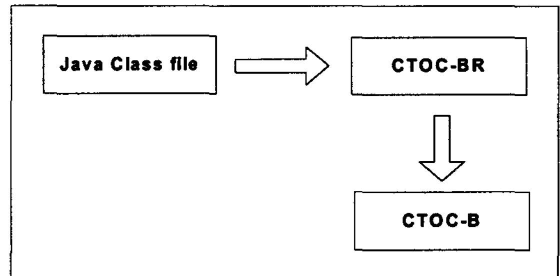
[그림2-1] 자바프로그램에서 바이트코드로 표현

이러한 자바 바이트코드는 스택기반의 코드로써 자바 가상머신에서 동작하고 그 결과로 생성된 스택코드는 추가적인 어떠한 변화 없이도 스택 머신에서 수행이 가능하기 때문에 생성된 코드에 대한 변환이 필요 없다는 장점이 있다. 하지만 바이트코드는 200개가 넘는 명령어로 구성되어 있어 분석이 힘들고 표현이 명확하지 않다는 단점이 있다. [그림2-1]은 자바 프로그램을 바이트코드로 나타낸 것이다.



[그림 2-2] CTOC의 내부구성도

[그림2-1]에서 바이트 코드는 스택 명령어가 여러 조각으로 나누어져 코드가 복잡하게 되어 분석이 어렵다. 또한 [그림2-1]에서 invokevirtual, getstatic등의 명령어들 뒤에 #인덱스가 나타나있는 것을 볼 수 있다. #인덱스는 상수 풀의 내용표현하고 있는 것으로 바이트코드는 상수 풀의 내용 없이는 분석이 용이 하지 못하다.



[그림 2-3] CTOC-BR의 구성도

[그림 2-2]은 CTOC의 전체적인 내부구성을 보여주고 있다. CTOC를 살펴보면 [그림 2-3]과 같이 자바 클래스 파일을 입력 받아 CTOC-BR(Class To Optimized Class Bytecode Translator)이라는 변환기에 의해 CTOC-B가 만들어 지고 이 만들어진 CTOC-B는 CTOC-TR(Class To Optimized Classes-Three address code tRanslator)이라는 3-주소 형태를 변환하는 변환기 CTOC-TR에 의해 CTOC-T가 만들어 진다. CTOC-T는 분석과 변환이 용이하기 때문에 최적화를 위해서 변환한다.

CTOC-RB (Class To Optimizer Classes-Return Bytecode)은 3주소 형태의 CTOC-T를 CTOC-B로 변환하는 부분이다. CTOC-RC(Class To Optimizer Classes -Return ClassFile)은 CTOC-B를 클래스파일로 변환시키는 부분이다.

CTOC-B는 바이트코드에서 3-주소 형태인 CTOC-T로 변환하기 위한 중간표현으로써 바이트코드보다 분석이 용이한 형태이다.

3. CTOC-B의 변환

자바 바이트코드에서 CTOC-B의 변환은 심볼 테이블을 이용한 패턴매칭 기법을 사용한다. CTOC-B에서는 바이트코드 명령어에 타입을 줌으로써 보다 분석이 쉽게 설계했다. 뿐만 아니라 CTOC-B에서의 타입화로 3-주소 형태인 CTOC-T에서 필요한 타입추론도 할 수 있게 된다.

[표3-1] 바이트코드 명령어와 CTOC-B 명령어

Bytecode 명령어	CTOC-B 명령어
iconst_0,iconst_1...	push *constant*
bipush,sipush	
ldc,ldc_w...	
iload,iload_0,lload_0...	load.*t local*
istore,istore_0lstore_0...	store.*t local*
iadd,ladd...	add.*t*
imul,lmul...	mul.*t*
ior,lor	or.*t*

[표 3-1]는 바이트코드 명령어와 CTOC-B의 명령어를 비교한 것이다. const, ldc, bipush등과 같이 표현이 비슷한 명령어들은 push 라는 명령어로 변환을 하고, iadd, iload, istore등은 add.t, load.t, store.t로 뒤에 타입을 주는 형태로 변환을 한다. 이러한 타입을 할당함으로써 바이트코드에서 명확하게 표현되지 않았던 타입에 대해 CTOC-B에서는 명확하게 사용된다. 또한 CTOC-B는 타입화한 명령어에 지역변수 명을 타입 뒤에 나타내줌으로써 바이트코드에서 3-주소형태의 CTOC-T로의 변환을 보다 용이하게 할 수 있다.

[표 3-2]에서는 [그림2-1]의 자바 프로그램을 CTOC-B코드로 변환을 나타내고 있다. 자바 바이트코드에서 CTOC-B로 변환에서 명령어뿐만 아니라 몇몇 형태적으로도 변화가 생기게 되는데 그 변화들에 대해 살펴보면, 우선 앞에서도 언급했듯이 바이트코드에서 필드, 메서드, 클래스, 상수의 접근을 하기 위해 사용했던 상수 풀을 바이트코드에서는 인덱스로 표현하고 있어 상수 풀 없이는 분석이 용이 하지 않았다. 따라서 CTOC-B에서는 상수 풀의 내용을 내부적으로 직접 표현함으로써 바이트코드에서 상수 풀의 내용 없이 분석이 용이하지 못하는 부분을 CTOC-B에서는 상수 풀의 내용이 없이도 분석을 쉽게 할 수 있다.

예를 들면

bytecode : ldc #37

CTOC-B : push ["Bad Argument"]

와 같이 직접적으로 표현을 한다. 상수 풀의 직접적인 표현은 CTOC-T로 변환할 때 상수 풀의 내용을 참조하지 않고도 코드의 조작을 쉽게 할 수 있다.

[표3-2] 자바 바이트코드에서 CTOC-B코드 변환

바이트코드	CTOC-B
0: iload_0 1: ifge 14 4: getstatic #25 7: ldc #37 9: invokevirtual #40 12: iconst_m1 13: ireturn	load.i x ifge [label 14] getstatic [java.lang.System.out] push ["Bad Argument"] invokevirtual [java.io.PrintStream.println] push -1 return.i
14: iload_0 15: iconst_5 16: if_icmpgt 23 19: iload_0 20: iload_0 21: imul 22: ireturn	[label 14] load.i x push 5 ificmpgt [label 23] load.i x load.i x mul.i return.i
23: iload_0 24: iconst_5 25: imul 26: bipush 16 28: iadd 29: ireturn	[label 23] load.i x push 5 mul.i push 16 add.i return.i

다른 형태상의 특징은 자바 바이트코드 명령어중 ifeq, ifcmpeq, goto와 같이 다른 명령어로 분기가 되는 명령어들에 대해서는 label 인덱스로 분기를 표현함으로써 복잡해질 수 있는 코드의 분석을 쉽게 한다.

CTOC-B는 자바 바이트코드에서 분석과 변환이 어려운 부분을 명시적으로 또는 직접적으로 표현함

으로써 자바 바이트코드에서 3-주소 형태로의 변환을 쉽게 하기 위한 중간 표현이다. 따라서 CTOC-B에서는 변환만을 하고 특별한 최적화 기법은 들어가지 않는다. 이후 CTOC-B는 CTOC-TR을 거친 후 3-주소 형태인 CTOC-T로 변환이 다시 이루어지고 CTOC-T에서 최적화 기법들이 적용되어 최적화를 이루게 된다.

4. 결론 및 향후 연구

자바의 실행 속도를 빠르게 하기 위한 방법들은 많이들 제시되었다. 본 논문에서는 이런 실행속도를 개선 하기위해 바이트코드를 직접적으로 변환하여 분석 간단한 CTOC-B를 설계하였다.

CTOC-B는 바이트코드에서 3-주소로 변환에서 바이트코드 보다 코드의 조작과 변환이 용이하게 만든 중간표현이다.

CTOC-B에서는 최적화에 기법들이 사용되지 않고 단지 변환만을 하였다. 향후 연구과제는 바이트코드에서 CTOC-B로 변환할 때 바이트코드 최적화 기법을 사용하여 좀 더 빠른 CTOC-B 코드의 생성할 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] Frank Yellin, "*The JIT Compiler API*", http://java.sun.com/docs/jit_interface.html, 1996

[5] Ronald Veldema,"*JCC, a native Java compiler*", Technical report, 1998

[6] A. Krall and R. Grafl, "*CACAO - A 64 bit Java VM Just-in-time Compiler*", Appeared at PPoPP'97 Workshop on Java for Science and Engineering Computation, 1997