제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

스택기반코드에서 효율적인 3-주소 코드로의 변환기 설계

김지민, 김기태, 조선문, 유원희
인하대학교 컴퓨터공학과
jimin9441@empal.com

Design of Translator for Efficient 3-Address Code from Stack Based Code

Ji-Min Kim, Ki-Tae Kim, Sun-Moon Jo, Weon-Hee Yoo
Dept. of Computer Science and Engineering, Inha University

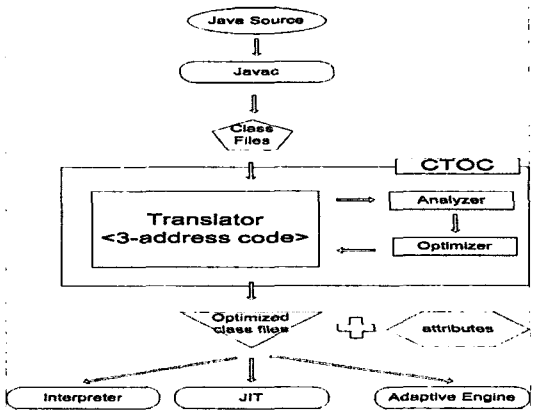
요 약

자바 언어는 객체지향 언어로써 인터프리터에 의하여 실행되고 구조 중립적이다. 자바 언어는 인터프린트 과정을 거치므로 다른 언어에 비해서 실행이 느리다는 단점을 가지고 있다. 자바 바이트코드의 실행 비용을 줄이기 위한 연구의 일환으로 본 논문에서는 자바 바이트코드 최적화기인 CTOC 중에서 스택기반 코드를 받아 들여 스택을 사용하지 않는 3-주소 코드로 변화시키는 CTOC-TR의 설계에 대하여 논한다. CTOC-TR은 총 3단계를 변환 과정을 수행하는데 첫 단계에서는 타입 없는 3-주소코드를 생성한다. 두 번째 단계에서는 스택변수와 지역변수를 나누는 과정을 수행하는데 이 과정은 타입을 정해주기 위해서 꼭 필요한 과정이다. 마지막으로 타입 추론 함수를 이용하여 나누어진 변수에 타입을 지정한다. 그 후 생성된 3-주소 코드를 분석기와 최적화기에 입력시켜 효율적인 3-주소 코드를 생성한다.

1. 서론

자바 언어는 객체지향 언어로써 인터프리터에 의하여 실행되고 구조 중립적이다. 또한 동적이고 다중스레드를 제공한다. 자바 언어는 컴파일 되면서 자바 바이트코드라는 중간 코드로 변환이 되고 그것을 자바가상기계(JVM: Java Virtual Machine)에 의하여 실행 한다. 자바는 자바가상기계에 기반을 두고 있으므로 특정 하드웨어나 운영체제에 영향을 받지 않고 실행이 가능하다. 그러나 자바 언어는 인터프린트 과정을 거치므로 다른 언어에 비해서 실행이 느리다는 단점을 가지고 있다.

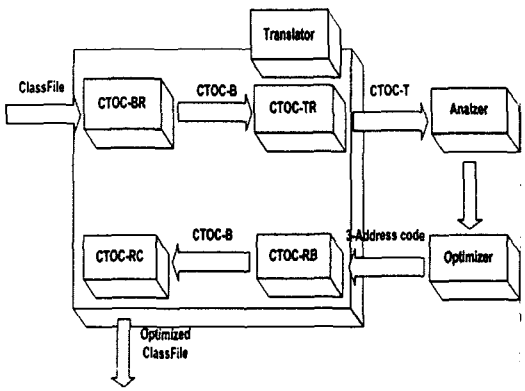
자바 바이트코드의 형태는 어셈블리 언어와 비슷하고 스택 접근 명령어를 정의하고 있다. 자바 가상기계는 인터프리터 방식을 이용하기 때문에 코드의 크기가 커질수록 실행에 많은 비용이 들어가기 때문에 이를 줄이기 위한 많은 연구가 진행 중이다. 자바 바이트코드의 실행 비용을 줄이기 위한 연구의 일환으로 본 논문에서는 자바 바이트코드 최적화기인 CTOC(Class To Optimize Class)에 대하여 설명하겠다. CTOC란 자바 바이트코드 최적화기로 CTOC의 변환단계는 크게 4단계의 변환기와 분석기, 최적화기로 구성되어 있다. [그림1]은 CTOC의 전체 구성도를 보여주고 있다.



[그림 1] CTOC의 전체 구성도습

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

첫 단계에서는 CTOC-BR(Class To Optimized Class- Bytecode tRanslator)을 이용하여 바이트코드를 CTOC-B(Class To Optimize○ Class-Bytecode)로 변환시키는데 CTOC-B란 형태는 바이트코드와 비슷하면서 스택기반의 타입이 명확한 중간언어이다. 두 번째 단계에서는 변환된 CTOC-B를 다시 CTOC-TR(Class To Optimized Class-Three address code tRanslator)에 넣어 스택을 사용하지 않는 3-주소형태의 중간표현인 CTOC-T(Class To Optimize Class-Three address code)로 변화시킨다. 그리고 CTOC-T에 핍홀 최적화 기법(*Peephole Optimization*)을 적용하여 변환된 코드를 최적화 한다. 세 번째 단계에서는 CTOC-RB(Class To Optimized Class -Return Bytecode)를 이용하여 3-주소형태의 코드를 다시 스택기반의 바이트 코드로 변환을 한다. 그리고 마지막으로 CTOC-RC(Class To Optimized Class -Return Classes)를 이용하여 클래스 파일로 변환한다. 이러한 과정을 통하여 최적화된 클래스 파일을 얻음으로써 자바언어의 단점인 실행 비용이 크다는 단점을 줄이는 것이 본 연구의 목적이다. 아래 [그림 2]는 CTOC의 내부 구성도이다.



[그림 2] CTOC의 내부 구성도

2. 관련 연구

1.최적화기 및 최적화 기법

실행시간에 드는 비용을 줄이기 위한 방법으로 JIT컴파일러[1], 네이티브 코드로 변환 방식, 그리고 바이트코드 최적화 방식 등이 있다. JIT컴파일 방식은 실행시간에 메소드 단위로 분석하여 다시 실행할 만한 메소드를 네이티브코드 형태로 생성하여 가지고 있다가 그것을 분석하여 네이트브 코드 형태로 저장하고 실행하는 방식을 의미한다. 네이트브 코드로 변환을 이용하는 방식은 JCC[2], Tobe[3], CACAO[4]방식 등이 있다.

바이트코드 최적화 방식에는 크게 전역 최적화와 지역 최적화방식을 이용한다. 자바 바이트코드에 대한 최적화는 기계독립적인 최적화 방법으로 자바 컴파일러가 생성한 비효율적이 코드에 대해서 지역 최적화나 기법들을 적용하여 효율적인 코드로 개선하는 것이다. 좀 더 세분화된 바이트코드에 의존적인 최적화 기법으로는 변수할당, dup를 이용한 최적화 , 네이티브 메소드 호출 등을 들 수 있다. 우선 변수할당 기법이란 일반적으로 자바 바이트코드에서 처음 4개의 변수를 참조할 때, 메소드 안에 있는 앞쪽의 4개의 변수는 1바이트의 크기만을 갖는 명령어를 사용하여 접근된다는 특징을 이용하여 프로그램에서 자주 사용되는 변수를 4번째 안에 선언을 하는 방식을 의미한다. dup를 사용하는 최적화에서 dup는 중복을 수행하는 바이트코드 명령어인데, 바이트코드는 스택 기반 언어이고, 자바 컴파일러는 확실한 상황에서만 중복 명령어를 생성하기 때문에, 이들 명령어를 좀 더 다양한 상황에 적용함으로써, 클래스의 코드의 양을 줄이고 성능의 향상을 가져 올 수 있게된다. 네이트브 코드 호출은 자바 바이트코드를 특별한 목적의 최적화된 네이트브 메소드들로 대체하는 방식을 의미한다. 특별한 코드의 유형에 대한 정보를 갖는 각 메소드의 테이블을 유지함으로써 복잡한 연산을 하는 코드를 네이트브 메소드 호출로 변화시켜주는 것이다. 하지만 단순한 코드에 대한 변환은 오히려 잦은 메소드 호출로 인한 오버헤드가 더 클 수 있다.

2.바이트코드 최적화와 분석의 문제점

바이트코드의 최적화의 문제점은 코드의 단편화, 타입정보의 상실, 그리고 스택 접근 명령어가 프로그램의 분석을 어렵게 만든다. 코드의 단편화 문제점은 복잡한 수식의 경우를 말한다. 다음 [그림3]에 나타나 있다. [그림3]를 보면 바이트코드에서 자바소스에서 int타입으로 선언되었던 변수들이 타입을 상실하는 것을 볼 수 있다. 이러한 변화는 분석을 어렵게 하는 이유 중의 하나이다. 그리고 [그림3]에서 바이트코드를 보면 모든 값들은 스택에 넣어서 연산을 한다. 따라서 스택 접근 연산들이 많아지게 된다. 또한 바이트코드 최적화기법은 기존의 3주소형태의 최적화 기법들을 적용하기 힘들다는 단점을 가지게

된다.

	0: iconst_1 1: istore_1 2: iconst_1 3: istore_2 4: iload_1 5: ifeq 17 8: iload_1 9: iload_2 10: iconst_3

int x=1, y=1;
 	
if (x!=0) x+=y*3;
else y=y*2;
	11: imul
	12: iadd
	13: istore_1
	14: goto 21
	17: iload_2
	18: iconst_2
	19: imul
	20: istore_2
	21: return

(a) 자바 소스	(b) 바이트코드

[그림 3] 코드단편화 예제

3. 스택기반의 바이트코드에서 스택이 없는 효율적인 3주소 코드로의 변환기 설계

3.1. CTOC-B

[그림 1]에서 바이트코드를 CTOC-BR에 입력하면 CTOC-B코드가 생성된다. CTOC-B는 스택 기반의 중간 언어로써 분석과 최적화를 위해 필요한 형태이다. CTOC-B의 형태는 바이트 코드와 유사한 니모닉 형태를 가지고 있다. 그러나 CTOC-B는 명시적 타입을 가지고 있다. CTOC-B의 일반적인 형태는 (add.i, add.l, add.f, add.d) 명령어.타입의 형태를 가지고 있다.

바이트 코드에서 COTC-B의 과정을 거치는 이유는 다음과 같다. 바이트코드의 타입은 암시적인 타입인데 반하여 CTOC-B의 타입은 명시적인 타입이다. 바이트코드는 상수 풀에서 정보를 가지고 오지만 CTOC-B는 상수 풀을 사용하지 않는다. 또한 CTOC-B는 사용되는 지역변수 이름을 명확하게 나타내어 준다. 이러한 관점에서 CTOC-B는 기존의 바이트 코드보다 판독성을 높이고, 분석을 용이하게 해준다. 또한 스택을 사용하지 않는 3-주소 코드 형식의 CTOC-T에서 사용된 스택 변수의 개수를 알기 위해서는 각 메소드마다 스택 최대 높이를 알아야 하기 때문에 CTOC-B로 변환을 해야 한다. 아래 [그림 4] 는 바이트코드와 CTOC-B의 간략한 예제로써 바이트코드와 CTOC-B와의 차이점을 보여주고 있다.

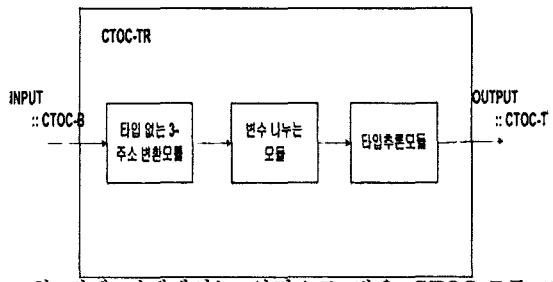
0: iconst_0 1: istore_2 2: iconst_1 3: istore_3 4: bipush 10 6: newarray int 8: astore 4 . . 80: iload_1 81: bipush 10 83: ifeq 15 86: return	push 0; store.i l1; push 1; store.i l2; push 10; newarray; store.a l3; . . label5: load.i l1; push 10; ifeq label0; return;
자바 바이트코드	CTOC-B 코드

[그림 4] 바이트코드와 CTOC-B의 형태변환

3.2. CTOC-T

3-주소형식의 중간 언어인 CTOC-T는 COTC-B를 CTOC-TR에서 입력으로 받아 변환된 결과로 나온 코드이다. CTOC-T는 스택을 사용하지 않는 3-주소 코드로써 3단계의 변환과정을 거치게 된다. 아래 [그림 5]는 CTOC-TR의 내부구조를 모듈별로 나타내고 있다.

[그림 5] CTOC-TR의 내부 구성도



첫 번째 단계에서는 입력으로 받은 CTOC-B를 타입이 없는 3-주소 형태로 변환시킨다. 이때 스택을 대신할 스택 변수를 사용하는데 형태는 $Sn(여기서 n이란 그 명령어에서 동작한 스택의 높이다.)이다. 앞에서 말했듯이 n의 최대 값은 사용된 스택의 최대 값과 같다. 변화를 위한 매핑 테이블이 존재하는데 이 테이블에 의하여 입력된 명령어는 패턴 매칭 기

법으로 변환이 된다. 아래 [그림 6]은 간단한 패턴 테이블을 나타내고 있다.

[그림 6] 간략한 패턴 테이블

변환전	변환후
unknow $S0, $S1;	unknow $S0#0, $S0#1, $S1#0, $S1#1;
$S0 = 1;	$S0#0 = 1;
$S1 = 2;	$S1#0 = 2;
$S0 = $S0 + $S1;	$S0#0 = $S0#0+$S1#0;
$S0 = "string";	$S0#1 = "string";
$S1 = 0.001;	$S1#1 = 0.001;

그러나 이렇게 생성된 스택 변수들은 문제점을 가지고 있다. 왜냐하면 스택이라는 공간은 하나의 스택에 여러 가지 타입이 올수 있기 때문이다. 예를 들어 1번 스택에 처음에는 int형의 자료가 들어오고 그 후에 float형이 들어와도 첫 번째 변환에서는 $S1이라는 같은 변수로 변환이 된다. 이러한 타입에 대한 문제를 해결하기 위하여 두 번째 변환단계에서는 각각의 스택변수를 사용된 순서 별로 나누게 된다. 아래 [그림 7]은 간단한 변환의 예이다.

[그림 7] 스택변수를 나누기

CTOC-B	CTOC-T
load.i x ;	$Sn = x ;
store.i l2 ;	l2 = $Sn ;
push 5 ;	$Sn = 5;
inc.i l3 ;	l3 = l3 + 1;
load.i x ; load.i y ; add.i ;	$Sn = x; $Sn+1 = y; $Sn = $Sn+1 + $Sn ;

그러나 여기서 단순이 번호로 나누는 것은 별의미가 없다. 왜냐하면 프로그램 제어에 영향을 주는 반복문이나 선택문의 경우에 다음 스택 변수를 어떻게 결정하는지가 가장 중요하다. 그래서 두 번째 단계에서는 반복문이나 선택문이 나타나는 경우에 대하여 SSA형태[5]를 이용했다. 기존에 방법은 대부분 사용-정의 고리를 사용하는데 SSA를 이용하게 되면 필요한 부분에 따라서 선택적으로 다음 변수를 택할 수 있다. 세 번째 마지막 단계로는 타입을 추론하는 단계로써 타입함수 T를 이용하여 타입을 추론한다. 두 번째 단계까지는 타입이 존재 하지 않았다. 이유는 두 번째 단계를 지나기 전까지는 스택 변수에 대한 명확한 변환이 이루어지지 않았기 때문이다. 타입함수 T는 배정문 처럼 변수에 대한 명확한 타입을 유추할 수 있는 문장에 의하여 계산된다. 그리고 타입이 명확하게 추론된 변수가 사용된 배정문을 찾아서 다음 변수의 타입을 추론하게 된다.

아래 [그림 8]은 CTOC-TR의 변화과정을 마친 후의 모습이다.

[그림 8] CTOC-TR 변화과정후의 모습

변환전	변환후
unknow $S0#0, $S0#1, $S1#0, $S1#1;	int $S0#0, $S1#0; String $S0#0; float $S1#1;
$S0#0 = 1;	$S0#0 = 1;
$S1#0 = 2;	$S1#0 = 2;
$S0#0 = $S0#0+$S1#0;	$S0#0 = $S0#0+$S1#0;
$S0#1 = "string";	$S0#1 = "string";
$S1#1 = 0.001;	$S1#1 = 0.001;

4.결론 및 향후 연구

본 논문에서는 스택기반의 바이트코드에 대한 가장 큰 문제점으로 실행속도가 느리다는 점에 착안하여 스택기반이 아닌 3주소형태 코드로의 변형과 생성된 코드에 대한 최적화를 실행하는 변환기 설계에 대하여 기술하였다.

향후 연구과제로는 최적화 부분에 대한 적용 알고리즘의 확립과 아직 기술하지 않은 3주소형태에서 다시 스택기반 코드로의 변화 과정 그리고 변환기에 대한 구현을 수행 할 것이다.

참고문헌

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

[2] Ronald Veldema, "JCC a native Java compiler", Technicaal report, 1998.

[3] Todd A. Proebsting, Greg Townsend, Patrick Bridges, "Toba: Java For Application A Way Ahead of Time(WAT) Compiler", COOT97, pp. 41-53, 1997

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

[5] Stenve S. Muchinck, "Advanced Compiler Design and Implementation", Mogan Kaufmann, 1997