스택 기반 코드에서 3-주소형태코드 생성을 위한 변환기 설계

김지민*, 김영국, 조선문, 김기태, 유원희
인하대학교 컴퓨터 정보공학과
e-mail:jimin9441@empal.com

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

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

요 약

자바의 특징 중에 한 가지는 자바 가상 기계를 기반으로 하고 있게 때문에 특정한 하드웨어나 운영체제에 영향을 받지 않고 독립적으로 수행이 가능하다는 것이다. 하지만 자바 언어로 개발된 애플리케이션은 C나 C++등 다른 언어로 작성한 프로그램에 비하여 실행이 매우 느리다는 단점을 가지게 된다. 이는 자바 가상 기계 에서 바이트코드가 인터프리터 방식으로 사용되기 때문이다.

이러한 단점을 보안하기 위하여 여러 가지 최적화 기법이 적용되고 있다. 본 논문에서는 이러한 방법으로써 바이트코드를 3주소형태 코드로 변환하는 변환기 설계에 대해서 제안할 것이다. 바이트코드에서 스택을 사용하지 않는 3주소형태 코드로의 변환하기 위하여 크게 세 단계를 걸친다. 첫째, 스택에 대한 명백한 참조를 가진 타입화된 스택기반의 중간표현을 생성한다. 둘째, 생성된 코드에서 타입에 대한 정보를 추출하고 추출된 정보를 저장하는 기억장소를 할당하여 추출된 정보를 저장시킨다. 셋째, 스택을 대신할 타입이 없는 지역변수를 생성하여 각각의 변수에 알맞은 타입을 분배함으로써 타입화되고 명백한 3주소형태 코드를 생성한다. 이러한 방식으로 스택기반 언어에서 발생하는 문제점을 해결한다.

1. 서론

자바는 객체지향 언어로써 인터넷 및 분산 환경 시스템에서 효과적으로 사용되기 위하여 개발된 언어이다. 자바의 특징 중에 한 가지는 자바 가상 기계(JVM)를 기반으로 하고 있게 때문에 특정한 하드웨어나 운영체제에 영향을 받지 않고 독립적으로 수행이 가능하다는 것이다. 하지만 자바로 개발된 응용프로그램은 C나 C++등 다른 언어로 작성한 프로그램에 비하여 실행이 매우 느리다는 단점을 가진다. 이는 자바 가상 기계에서 바이트코드가 인터프린터 방식으로 번역되기 때문이다[1].

자바 가상 기계는 스택기반의 기계이다. 따라서 바이트코드 역시 스택에 대한 동작들로 정의되어 있다. 자바 바이트코드는 스택기반의 중간 표현을 이용해서 자바 가상 머신에서 동작한다. 스택기반의 코드는 즉시 이용이 가능하고, 결과로 생성된 스택 코드는 변환이 필요 없이 실행되는 장점들을 가지고 있다[2].

스택기반의 바이트 코드의 문제점은 명령어가 200여개가 넘고, 표현이 명백하지 않다는 문제점을 가지고 있다. 명확한 x = a + b 와 같은 3주소 형태의 코드가 아니 오퍼랜드 스택에 영향을 주는 iload, iadd, pop등 의 명령어로 표현되기 때문에 판독성의 저하를 초래하게 된다. 또한 적재(load)와 저장(store)의 불필요한 다중 사용 역시 분석과 판독을 어렵게 한다. 또한 바이트코드의 형태적 특징으로 원시코드가 가지고 있는 많은 정보에 대한 명확한 정보의 전달이 어렵고 이러한 전달과정에서 원시코드 정보의 내용을 상실하게 된다[3]. 무엇보다 자바의 가장 큰 단점은 실행 속도가 느리다는 것이다.

그러므로 자바 프로그램을 효율적으로 실행하기 위하여 최적화된 코드로의 변화는 필수적인 작업이다.

본 논문에서는 스택기반의 바이트코드를 최적화 시키는 프레임워크인 CTOC(Class To Optimized Class)에서 스택기반 코드를 스택이 없는 3주소 형태 코드로 변형 시키는 CTOC-TR(Class To Optimized Class- Three address code tRanslater)에 대하여 논할 것이다.

2. 관련연구

1. Bytecode

자바 바이트코드는 하드웨어나 운영체제에 비종속적인 특징을 가지고 있으며 자바 컴파일러(Javac)에 의하여 생성되는 중간언어이다. 또한 바이트코드는 스택기반의 언어로서, 크게 니모닉과 피연산자로 이루어진다. 니모닉은 동작에 관련된 명령이고 피연산자는 그것에 대한 정보를 제공한다[2]. 그리고 대부분의 수행된 결과 값은 다시 피연산자 스택에 저장된다. 이러한 동작의 반복은 필요 없는 적재와 저장을 유발하는 이유가 되기도 한다. 또한 바이트코드의 피연산자는 컴파일 시간에 결정되며 실행 시간에 실질적으로 계산되어 결과값이 스택에 저장되는 형태를 가진다.

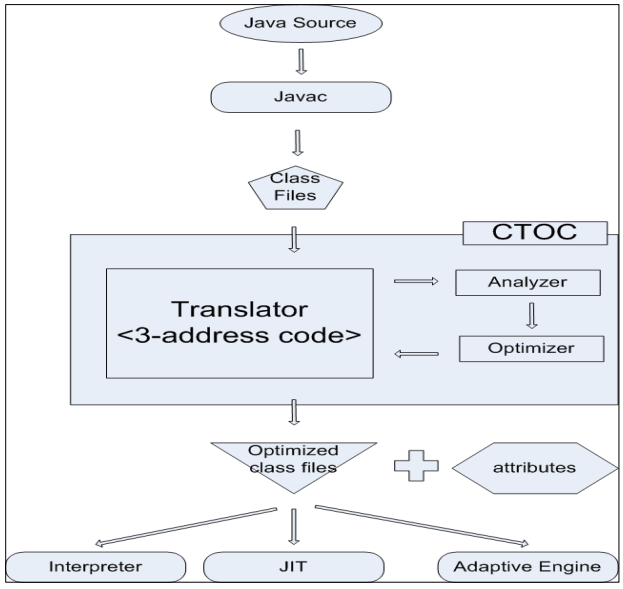
2. 바이트 코드 최적화 기법

바이트코드는 플랫폼에 비종속적으로 실행이 가능하지만 인터프리터방식으로 수행되기 때문에 수행속도가 매우 느리다. 만약 프로그래머가 작성한 프로그램의 크기가 크면 클수로 이러한 단점은 명확하게 들어나게 된다. 이러한 단점을 극복하기 위해서 최적화는 필수적이다[4].

바이트 코드 최적화 기술은 크게 상수의 사용 증가, 길이의 감소, 루프(loop)를 풀어서 기술하는 방법 등이 있다. 상수의 사용을 증가시킨다는 것은 만약 최적화 프로그램에서 해당 변수가 더 이상 바뀌지 않는다는 것이 확인된다면 변수를 상수로써 치환한다는 것이다. 길이를 감소시킨다는 것은 시간이 오래 걸리는 작업을 동일한 효과를 내면서 좀 더 짧게 끝날 수 있는 작업으로의 대체를 의미한다. 루프를 풀어서 기술한다는 것은 루프의 실행 시간보다 루프 내의 조건을 테스트하는데 더 많은 시간이 걸릴 경우 루프를 풀어서 수행함으로써 코드의 길이는 길어지지만 총 명령어의 개수는 줄임으로써 실제 실행 속도를 향상 시키는 것을 의미한다. 또한 루프의 카운터로써 사용되던 지역변수를 더 이상 쓸 필요가 없어지게 된다[5].

3. CTOC Overview

본 논문은 CTOC의 설계 및 개발의 일환으로 Javac에서 생성된 클래스파일을 CTOC에 내제된 CTOC-BR(Class To Optimized Class- Bytecode tRanslater)를 통하여 나온 CTOC-B(Class To Optimized Class- Bytecode)코드를 이용하여 스택기반의 코드를 스택을 사용하지 않는 3주소형태의 코드인 CTOC-T(Class To Optimized Class- Three address code)로 변형 시키는데 목적을 두고 있다. CTOC의 기본 개념은 아래의 [그림 1]에서 처럼 원시코드가 자바 컴파일러에 의하여 중간언어 형태인 바이트코드로 변화될 때 스택기반 언어인 바이트코드가 가지고 있는 문제점을 보안하고자 3주소형태의 코드로 변환 시킨 후 그것을 분석과 최적화를 통하여 다시 최적화된 바이트 코드형태로 변환하여 인터프리터, JIT컴파일러 등으로 실행한다.



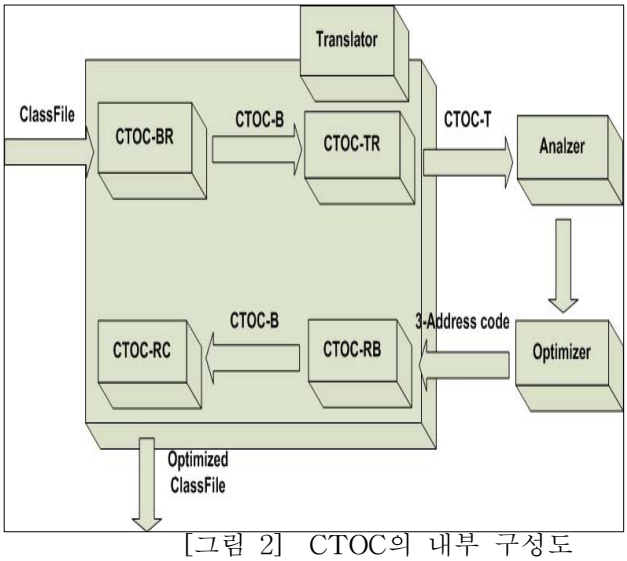
[그림 1] CTOC 설계 모습

CTOC는 기본적으로 크게 3가지 단계로 나누어 진행되는데 첫 번째 단계에서는 CTOC-B에 의하여 생성된 스택기반의 코드에 속성을 부여한다. 바이트코드의 묵시적 타입을 명백한 타입으로 변화시킴으로써 명령어의 개수를 줄이고 표현을 명확하게 함으로써 분석을 용이하게 한다. 두 번째 단계에서는 이러한 스택기반의 CTOC-B를 CTOC-TR를 이용하

여 3주소 형태로 변형하고 최적화를 적용시켜서 스택기반 언어의 문제점을 보안하고 최적화된 3주소 형태의 코드를 산출해낸다. 마지막으로는 최적화된 코드를 다시 스택기반의 코드로 변환하여 인터프리터나 JIT컴파일러 등으로 보내어 최적화된 코드를 수행하도록 한다.

3. 스택기반의 바이트코드에서 스택이 없는 3주소 코드로의 변환기 설계

CTOC-BR에 의하여 생성된 CTOC-B코드는 바이트 코드에 타입을 주고 정형화함으로써 바이트코드의 판독성과 분석을 용이하게 해주고 있다. 그러나 CTOC-B코드는 아직 스택기반의 중간 표현으로서 스택기반 언어들이 가지고 있는 많은 명령어, 명백하지 못한 표현, 적재와 저장의 불필요한 다중 사용 그리고 실행속도가 느리다는 문제점을 극복하지 못하고 있다. 서론에서 제기된 스택기반 언어의 문제점을 보안하기 위하여 설계된 CTOC-TR은 스택기반의 CTOC-B에서 스택을 제거하기 위해 스택 슬롯에 대한 묵시적 참조를 스택 슬롯을 대신할 임시변수에 대한 명백한 참조로 변환시켜 스택을 사용하지 않는 3주소 코드형태로 변환을 시키는데 중점을 두고 있다.



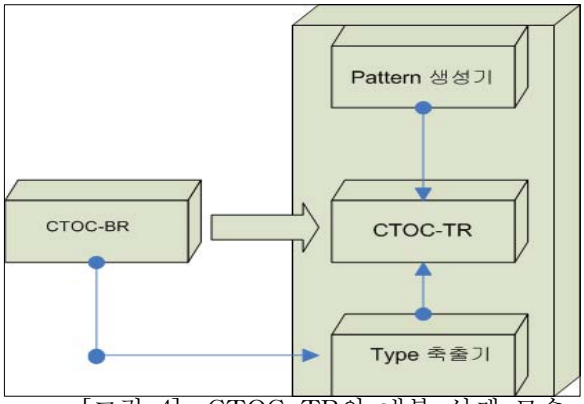
[그림 2] CTOC의 내부 구성도

[그림 2]의 CTOC-B코드에서 CTOC-T코드로의 변환은 기본적으로 3가지 과정을 걸쳐서 이루어진다. 첫 번째 과정은 생성된 CTOC-B코드를 동일한 의미를 갖는 3주소 형태로 변화시키면서 스택을 대신할 타입이 없는 지역변수를 생성하는 것이다. 이때 CTOC-B코드는 스택의 높이에 대한 정보를 가지고 있어야 한다. 이유는 스택 높이에 대한 정보를 이용하여 스택을 대신할 지역변수 개수에 대한 정보를 추출할 수 있기 때문이다. 이러한 과정에서 최대 스택 높이만큼의 슬롯은 모두 지역변수로 변환된다. 스택의 최대 높이에 대한 정보는 깊이 우선 탐색을 통하여 이루어지고 이것에 대한 보증은 자바 가상기계 명세에 나타나 있다[6]. 스택 슬롯을 대체할 변수를 표기하기 위하여 생성된 지역변수의 앞에는 “&”기호를 붙임으로 기존의 지역변수와 구분을 한다. [그림 3]은 코드의 변환을 보여주는 간단한 예이다.

var x, y, z	untyped &s0, &s1, x, y, z
I.load x	&s0 = x
I.load y	&s1 = y
I.add	&s0 = &s0 + &s1
I.store z	z= &s0
<CTOC-B>	<CTOC-T>

[그림 3] 변환 전, 후의 모습

두 번째 단계에서는 각 지역변수에 타입을 지정한다. 스택기반의 언어는 스택을 이용할 때, 타입에 대한 제약이 없이 데이터에 대한 참조가 가능 했지만 지역변수로 나타내게 된다면 타입에 대한 명백한 표기가 존재해야만 한다. 그러므로 각 스택 슬롯에 데이터에 대한 참조가 몇 번 있었는지 또 몇 번째로 되었는지에 대한 계산이 되어야 한다. 그리고 참조에 대한 카운터에 각각의 타입정보를 삽입함으로써 타입에 대한 정보를 저장한다. 이렇게 축출된 정보를 이용하여 각 슬롯마다 사용된 횟수와 순서에 맞게 생성된 스택 변수에 타입을 할당해준다.



[그림 4] CTOC-TR의 내부 설계 모습

[그림 4]는 CTOC-TR의 내부 구성도를 보여주고 있다. CTOC-BR에서 추출한 바이트코드의 타입정보를 기반으로 CTOC-TR의 타입 추출기에서 타입을 추출해 낸다. 패턴 생성기는 패턴을 입력시키면 패턴에 알맞은 패턴테이블을 생성하고 CTOC-B에 대한 매칭을 시작한다. 이러한 작업을 통하여 스택이 없는 3주소형태의 코드가 생성이 된다.

지역변수가 타입을 가지게 된다면 이러한 결과로 3주소형태의 코드에서는 불필요하게 만든 코드들이 발생하게 된다. 왜냐하면 스택기반의 언어에서는 적재와 저장의 반복으로 인한 불필요한 동작들이 기술되어 있기 때문이다. 그러므로 세 번째 단계에서는 3주소형태의 코드의 형태상 불필요한 동작에 대한 명령 문장에 대한 삭제를 수행해야한다. 이러한 불필요한 문장의 제거는 복사 전파와 상수 전파를 이용하여 제거가 가능하다. 하지만 여기서 문제점은 전방 전파(forward propagate)뿐만 아니라 후방 전파(backward propagate)의 혼합된 전파를 통해서만 이러한 코드 삭제가 가능하다는 것이다.

4.결론 및 연구방향

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] 고광만, “바이트코드로부터 네이티브 코드 생성을 위한 중간 코드 변환기의 설계 및 구현”, 멀티미디어학회, 멀티미디어학회 논문지 Vol.5, No.3, pp342-250, 2002

[2] John Meyer, Troy Downing, "Java virtual Machine", O'RELLAY, 1997

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

[4] 홍경표, 이야리, 오세만, “자바 클래스 파일 최적화”, 한국정보과학회, ‘01 봄 학술발표논문집(A), 제 28권 1호, pp.55-557, 2001

[5] Joshua Engel 저/곽용재 역 “자바 가상 머신 프로그래밍” 인포북

[6]Tim Lindhoim, Frank Yellin, "The Java Virtual Machine Specification second edition", http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html,1999