제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

CTOC에서 중간 코드에서 효율적인 바이트코드로의 변환기 설계

김영국, 조선문, 김기태, 유원희
인하대학교 컴퓨터 정보공학과
e-mail:odin004@hotmail.com

Design of Translator for Efficient Bytecodes from Intermediated Codes in CTOC

Young-Kook Kim, Sun-Moon Jo, Ki-Tae Kim, Weon-Hee Yoo
Department Computer Science & Information Inha University

요 약

자바는 자바 가상기계를 사용해서 플랫폼에 독립적으로 사용할 수 있는 장점을 가진다. 그러나 자바 가상머신은 인터프리터 방식을 이용하기 때문에 다른 컴파일언어에 비해서 수행속도가 느리다는 단점을 가진다. 이런 단점을 극복하기 위해서 많은 최적화나 JIT컴파일러 그리고 네이티브 코드로의 변환과 같은 연구들이 많이 연구되었다. 이전의 연구들은 한계성을 가지고 있어서 자바에서 수행시간의 문제는 지금도 문제로 남아있다.

이전의 문제점 해결의 극복의 대안으로 바이트코드를 최적화하는 프레임워크인 CTOC를 설명하고, CTOC에서 사용하는 중간코드로 사용하는 3-주소형식의 CTOC-T를 바이트코드로 변환하고, CTOC-T에서 바이트코드형식으로 변환할 때 생기는 과다한 load/store의 문제점을 해결한다.

1. 서론

자바는 가상기계를 이용하여 플랫폼에 독립적이라는 장점을 가진 언어이다[1]. 이러한 장점으로 인하여 프로그램 수행속도가 느리다는 문제점을 가진다. 문제점을 해결하기 위한 연구로 바이트코드 최적화[2, 3, 4]나 JIT컴파일러[5] 그리고 네이티브 코드로의 변환[6, 7]등의 연구가 이루어졌다. 우선 바이트코드 최적화의 경우에는 바이트코드의 분석과 변환의 어려움으로 인하여 한계성을 가지고, JIT컴파일러의 경우에는 첫 실행시에 분석을 통해서 다시 실행될만한 메소드만을 최적화된 네이티브코드를 생성하기 때문에 한번의 실행만을 요구하거나 최적화된 네이티브코드를 생성하지 않은 경우에는 수행속도의 문제점이 해결되지 않고, 네이티브 코드로의 변환의 경우에는 바이트코드를 직접 변환하기 때문에 변환된 네이티브 코드에대해서 최적화를 수행해야하는 문제점이 생긴다.

이런 문제점들을 극복하는 방안으로 바이트코드의 분석과 최적화의 어려움을 해결하기 위해서 분석과 최적화하기 쉬운 형태인 3주소형태의 중간코드를 가지는 바이트코드 프레임워크인 CTOC(Class To Optimized Classes)에서 중간코드에서 바이트코드로의 변환기에서 변환시 생기는 과다한 load/store의 문제점을 해결하기 위한 효율적인 변환기를 제안한다.

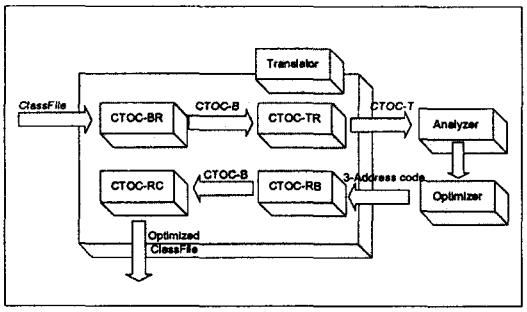
따라서 본 논문은 2장에서는 CTOC에 대해서 설명하고 3장에서는 본 논문에서 중간코드를 효율적인 바이트코드로의 변환기를 설계하고, 4장에서는 결론 및 향후 연구에 대해서 언급한다.

2. CTOC의 개요

자바 최적화 프레임워크는 CTOC-BR(Class To

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

Optimized Classes - Bytecode tRanslator), CTOC-TR(Class To Optimized Classes-Three address code tRanslator), CTOC-RB(Class To Optimized Classes-Return Bytecode), CTOC-RC(Class To Optimized Classes-Return Classfile)로 이루어진다. 전체의 구성도는 [그림 1]과 같다.



[그림 1] CTOC의 전체구성도

CTOC-BR(Class To Optimized Classes - Bytecode tRanslator)은 클래스파일에서 바이트코드를 추출한다. 바이트코드에서 타입을 나타내지 않는 명령어의 종류인 dup와 swap등의 타입이 없는 경우에는 타입 추론기를 이용하여 타입을 추론한다. 추론한 타입정보와 명령어에서 얻은 타입정보 그리고 상수풀에서 얻은 정보를 이용하여 CTOC-B를 생성해낸다. CTOC-B는 니모닉 코드에 타입을 붙여서 사용하기 때문에 바이트코드와 비슷한 모양을 갖는다. 그러나 필요한 상수풀의 정보를 포함하는 코드를 생성하기 때문에 더 이상의 상수풀이 필요로 하지 않는다. 따라서 CTOC-B는 상수풀을 포함하지 않는다. CTOC-B의 예를 들면 imul라는 바이트코드의 표현을 mul.i로 나타낼 수 있다.

CTOC-TR(Class To Optimized Classes-Three address code tRanslator)는 스택을 제거하기 위해서 클래스 파일에서 메소드 단위로 작성한 최대스택크기를 이용한다. 각각의 메소드 단위로 최대 스택 크기만큼 임시변수를 생성하고 임시 CTOC-T를 생성한다. 생성한 임시변수와 지역 변수들은 타입충돌을 피하기 위해서 타입에 따라 변수를 분리해서 코드를 생성한다. 생성된 코드는 불필요한 문장을 포함하기 때문에 데이터흐름분석을 이용해서 이러한 문장들을 제거한다.

Analyzer는 프로그램의 분석과 최적화 정보를 생성한다. Analyzer는 메소드 단위로 기본 블록을 생성하고 생성된 기본 블록 단위로 데이터 흐름 분석과 제어 흐름 분석을 통한 정보를 생성한다.

Optimizer는 3주소 형태의 최적화를 수행하는 기반과 최적화를 수행해준다. Optimizer에서 제공하는 최적화 기반을 이용하여 기존의 연구된 최적화방법을 조합하여 최적화를 수행하여 최적화된 3주소코드를 생성할 수 있다. 또한 새로운 최적화 알고리즘을 적용 가능한 자바로 구성된 API를 제공해준다.

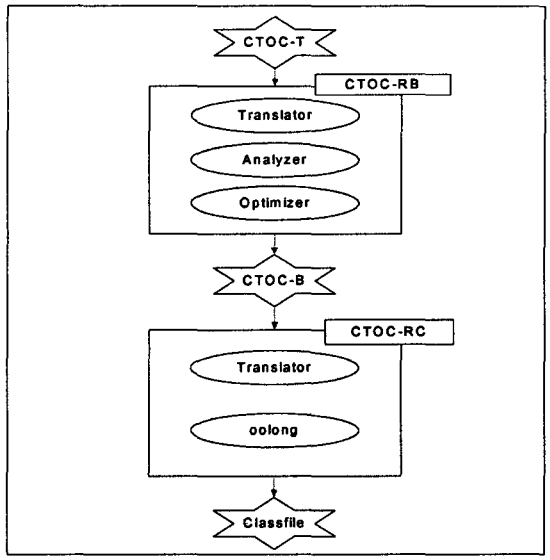
CTOC-RB(Class To Optimized Classes-Return Bytecode)와 CTOC-RC(Class To Optimized Classes-Return Classfile)은 안전한 실행을 보장하는 클래스파일로 변환하기 위해서 여러 단계를 거쳐서 변환하게 된다.

CTOC-RB는 우선 CTOC-T를 CTOC-B로 변환하기 위해서 AST(Abstract Syntax Tree)를 구성하고, 일반적인 트리 탐색기법인 전위 순회법(preorder traversal)을 이용하여 코드를 CTOC-B로 변환한다.

CTOC-RC는 CTOC-B를 oolong의 스택 기반 코드로 스트링 패턴 매칭 테이블을 이용하여 코드를 변환한다. 변환된 oolong형식의 코드를 oolong을 이용하여 클래스 파일을 생성한다[8].

3. CTOC-T를 효율적인 바이트코드로의 변환기의 설계

CTOC-T를 바이트코드로의 변환기의 구성도는 [그림 2]와 같다.



[그림 2] CTOC-T를 바이트코드로의 변환기의 구성도

우선 CTOC-T의 특징은 다음과 같다. 첫째 CTOC-T는 3주소 형식의 코드이므로 스택이 없다. 둘째 최대 피연산자의 수는 3주소 형식이므로 최대 2개를 가진다. 셋째 모든 변수는 명시적으로 타입을 가진 변수로 선언한다. 넷째 파라메터와 같은 특수한 변수는 특수한 규칙에 의하여 초기화해서 사용한다. CTOC-T의 예제로 HellowWrold에 대해서 [표 1]에 나타나 있다.

[표 1] CTOC변환 예제

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hellow World!");
    }
}
```

(a) source

```
public class HelloWorld extends java.lang.Object
{
    public static void main(java.lang.String[])
    {
        java.lang.String[] args;
        java.io.PrintStream $r0;
        args := @parameter0: java.lang.String[];
        $r0 = <java.lang.System: java.io.PrintStream out>;
        invokevirtual $r0.<java.io.PrintStream:
            void println(java.lang.String)>("Hellow World!");
        return;
    }
    public void <init>()
    {
        HelloWorld this;
        this := @this: HelloWorld;
        invokespecial this.<java.lang.Object: void <init>()>();
        return;
    }
}
```

(b) CTOC-T

CTOC-RB은 Translator, Analyzer, Optimizer로 구성된다. Translator는 CTOC-T를 CTOC-B로 변환한다. Analyzer는 CTOC-B에 대한 최적화 정보를 생성한다. Optimizer는 Optimizer는 3주소 코드에서 스택 기반 코드로 변환할 때 생기는 불필요한 load/store문은 3주소 코드 최적화가 이루어진 코드를 스택 기반 코드로 변환했을 경우에 원래의 바이트코드보다 실행이 저하되는 결과를 낳을 수도 있다. 이러한 문제점을 해결하기 위해서 불필요한 load/store문을 제거할 필요가 있다. 따라서 Optimizer는 불필요한 load/store문을 제거하는데 사용된다.

CTOC-RB에서 Translator는 CTOC-T 코드를 인식할 수 있는 Parser Tree를 생성한다. 생성된 Parse Tree를 AST(Abstract Syntax Tree)로 변환한다. AST에 대해서 전형적인 트리 탐색 기법인 전위 순회법(preorder traversal)을 이용하여 CTOC-B를 생성해낸다. 생성된 코드는 불필요한 load/store문을 가지고 있다. 따라서 불필요한 load/store문을 제거하기 위해서 코드를 CTOC-TR에서는 Analyzer를 통해서 분석을 한다.

CTOC-RB에서 Analyzer는 생성된 CTOC-B를 메소드 단위로 기본 블록을 생성한 후에 데이터 흐름 분석과 제어 흐름 분석을 통해서 정보를 생성한다. 생성된 정보는 기본 블록 단위로 저장한다.

CTOC-RB에서 Optimizer는 Analyzer에서 생성된 정보를 통하여 과다한 store/load에 대한 최적화를 적용한다. 최적화를 적용하기 위해서 실제로 많이 나타나는 store와 load에 대한 패턴을 정의하고 정의된 패턴에 대해서 코드를 변환한다. 정의된 패턴은 크게 store/load 패턴과 store/load/load 패턴으로 구분한다. store/load 패턴과 store/load/load 패턴은 Analyzer에서 데이터 흐름 분석에서 기본 블록 단위로 생성한 정보를 이용하게 된다. 기본 블록 내에 변수들의 값의 변화를 Analyzer가 생성한 정보를 이용하여 체크한다. 왜냐하면 변수들의 값의 변화가 없을 경우에만 위의 두 패턴이 적용 가능하기 때문이다.

우선 store/load패턴을 적용하기 위해서는 각각의 변수명마다 store위치와 load위치를 레이블을 이용하여 구분한다. 구분된 store와 load들 중에서 우선 가장 근접해 있는 구문들부터 store/load패턴을 적용한다. store문 다음에 바로 load문이 올 경우에는 두 개의 구문을 모두 삭제한다. 이와는 다르게 store와 load가 떨어져 있을 경우 store와 load를 삭제하고 dup와 swap을 이용하여 적용한다.

store/load/load의 패턴의 경우에는 패턴을 적용하기 위해서는 각각의 동일한 변수에 대해서 store문과 load문들을 레이블을 이용하여 구분한다. 구분된 store문과 load문들 중에서 가장 근접해 있는 구문들부터 store /load/load 패턴을 적용한다. store문 다음에 load/load구문이 올 경우에는 우선 store문과 load문을 제거하고 다음에 오는 load문에 위치에 dup문을 적용한다. 이에 대한 예제는 [표 2]에 잘 나타나 있다.

[표 2] CTOC-B의 최적화의 예

int k, x = 1; k = x+x;
(a) Source
push 1 store.i x load.i x load.i x iadd store.i k return
(b) CTOC-B
push 1; dup1.i; add.i; store.i k; return;
(c) 최적화된 CTOC-B

다음으로 CTOC-RC은 Translator와 oolong으로 구성된다. Translator는 CTOC-B코드를 oolong형식의 스택기반 코드로 변환한다. oolong형식으로 변환된 CTOC-B코드를 oolong을 이용하여 Classfile로 변환한다.

CTOC-RC에서 Translator는 CTOC-B코드를 oolong형식의 코드로 변환하기 위해서 스트링 패턴 매칭 기법을 사용한다. 스트링 패턴 매칭 기법을 사용하기 위해서 CTOC-B와 oolong의 문법에 대한 패턴 매칭 테이블을 구성한다. 구성된 패턴 매칭 테이블을 이용하여 oolong형식의 코드를 생성한다.

CTOC-RC에서 oolong은 CTOC-B에서 바이트코로의 변환은 스트링 패턴 매칭 기법으로 쉽게 변환할 수 있으나 바이너리 형식의 클래스파일을 생성하기에는 어렵기 때문에 사용한다. 따라서 바이너리 형식의 클래스파일을 생성하기 위해서 oolong형식으로 구성된 코드를 oolong을 이용하여 클래스파일을 생성한다.

4. 결론 및 향후 연구

자바는 플랫폼에 독립적인 자바는 가상기계를 사용함으로써 수행속도가 느리다는 문제점을 가졌다. 문제점의 해결 방법으로 연구되고 있는 최적화, JIT 컴파일러, 네이티비브 코드로의 변환등 방법등이 있으나, 이러한 해결 방법들도 한계성을 가지고 있었다. 최적화는 모든 상황을 고려할 수 없는 한계성을 가졌고, JIT는 첫 실행 시 분석을 통해서 다시 실행될만한 메소드만을 최적화된 네이티브코드를 생성하기 때문에 한번의 실행만을 요구하거나 최적화된 네이티브코드를 생성하지 않은 경우에는 수행속도의 문제점이 해결되지 않았다. 또한 네이티브 코드로의 변환은 단순히 네이티브코드로의 변환만 이루어졌다.

한계성을 해결하기 위해서 제시한 자바 프레임워크에서 중간언어인 CTOC-T에서 스택기반언어인 CTOC-B를 생성할 때 생기는 과다한 store문과 load문으로 해결하는 3주소 코드를 효율적인 스택기반 코드로의 변환기를 설계하였다. 향후 연구과제는 본 논문에서 제시하는 변환기를 구현하고, 효율적인 코드 생성을 위한 최적화 알고리즘을 적용한 모듈을 생성하는 것이다.

참고문헌

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

[2] Taiana Shpeisman, Mustafa Tikir, "Generating Efficient Stack Code for Java", Technical report, University of Maryland, 1999

[3] preEmptive Solutions, "DashO Whitepaper", http://preemptive.com/downloads/documentation.html, 2002-2004

[4] Geoff Cohen (Duke/IBM), Jeff Chase (Duke), David Kaminsky (IBM),"Automatic Program Transformation with JOIE", in Proceedings of the 1998 USENIX Annual Technical Symposium, 1998

[5] Frank Yellin, "The JIT Compiler API", http://java.sun.com/docs/jit_interface.html, 1996

[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

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

[8] Joshua Enagel저, 곽용재 역, "자바 가상머신 프로그래밍", 인포북, 2000