CTOC에서 3주소 코드를 효율적인 스택 기반 코드로의 변환기 설계

김영국, 김기태, 조선문, 김웅식, 유원희
인하대학교 컴퓨터 정보공학과
odin1004@hotmail.com

Design of Translator for Efficient Stack Based Codes from 3-Address Codes in CTOC

Youngkook Kim[0], Ki-tae Kim, Sunmoon Jo, Woongsik Kim. Weonhee Yoo
Department Computer Science & Information Inha University

요 약

자바는 객체 지향 언어이고, 분산 환경을 지원하고, 플랫폼에 독립적인 장점을 갖지만 다른 C나 C++같은 컴파일언어에 비해서는 실행속도가 느리다는 단점을 가진다. 이러한 단점을 극복하기 위한 방법으로 네이티브 코드로의 변환, 코드 최적화, JIT 컴파일 방법등을 이용한다. 그러나 이전 방법들은 다음과 같은 한계점을 가진다. 클래스 파일을 네이티브 코드로의 변환은 플랫폼의 종속되고, 코드 최적화 방법은 고유의 최적화 방법만을 적용할 수 있었고, JIT컴파일 방법은 한 번의 실행후 다음 실행해야 속도향상을 꾀할 수 있었다.

본 논문은 바이트 코드를 최적화하기 위한 자바최적화 프레임워크를 설명하고 자바최적화 프레임워크의 구성을 하는 부분중 3주소형식의 중간코드를 스택기반 코드로 변환하는 부분을 설계하고, 3주소 코드로 변환한 중간 코드를 스택 기반 코드로 변환하면서 생기는 과다한 load/store의 문제점을 지적하고 그것을 해결할 수 있는 변환기를 제안한다.

1. 서 론

자바는 객체지향언어이고, 분산 환경을 지원하고, 플랫폼 독립적인 특성을 갖는 언어이다. 이러한 장점을 가지고 있는 자바는 다른 C나 C++같은 네이티브 코드로 컴파일되는 언어에 비하여 실행속도 저하라는 단점을 가진다. 단점을 극복하기 위한 방법으로 다음과 같은 방법들이 있다.

첫째 클래스 파일을 네이티브 코드로 변환하면 프로그램의 실행속도는 C나 C++에 비하여 저하되지 않지만 플랫폼 독립의 장점을 사용할 수 없다는 단점을 가진다. 대표적인 툴로는 JCC[1], Toba[2], CACAO[3]가 있다. 둘째 최적화를 이용한 방법은 기존의 연구되었던 최적화 방법중에서 하나이상의 방법을 조합한 방법을 고정해서 사용하므로 항상 최적화가 되는 것만은 아니다. 대표적인 툴로는 Dash0[4],JOIE[5]이다. 셋째 JIT는 코드를 실행하면 JIT가 이 바이트 코드를 네이티브 코드로 컴파일하기 때문에, 나중에 동일한 메소드를 실행하면 다시 해석하지 않고 시스템 코드를 실행하므로 반드시 한번의 실행이 필요하다는 단점을 가진다. 대표적으로 Sun사의 JavaVM은 JIT를 포함하고 있다[6].

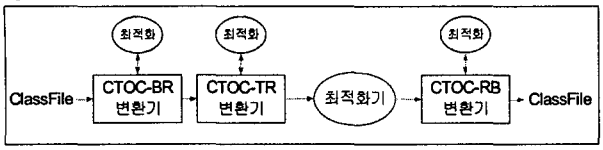
본 연구는 한국과학재단 목적기초연구(R05-2004-000-11694-0)지원으로 수행되었음.

자바의 실행시간에 대한 해결방안으로 제시되었던 방법들도 각각의 문제점을 가지고 있었다. 따라서 본 논문은 실행시간의 문제점의 해결방안으로 바이트 코드를 최적화하기 위한 자바 최적화 프레임워크를 소개하고 자바 최적화 프레임워크에서 사용되는 3주소 코드를 스택 기반 코드로 변환해주는 변환기를 제안한다. 그리고 변환기에서 나타날 수 있는 과다한 load와 store문제를 변환기내에서 최적화 패턴을 이용하는 방법과 데이터 흐름 분석을 이용한 해결방안을 제시하고 향후 연구 방향에 대해서 기술한다.

2. 자바최적화 프레임워크의 설계 개요

자바 최적화 프레임워크는 크게 4개의 부분으로 이루어진다. 첫 번째 클래스 파일를 분석하기 쉬운 CTOC-B(Class To Optimized Classes-Bytecode)로 변환하고 최적화를 수행하는 부분(CTOC-BR)이 있다. 두 번째 CTOC-B를 3주소 형식의 CTOC-T(Class To Optimized Classes-Three address code)로 변환하고 3주소 코드로 변환할 때 생기는 불필요한 부분을 최적화를 수행하는 부분(CTOC-TR)이 있다. 세 번째 부분은 CTOC-T의 3주소 코드 최적화 기법을 적용하는 부분(최적화기)이다. 마지막으로 3주소형의 CTOC-T의 최적화된 코드를 스택코드로 변환하는 부분(CTOC-RB)이 있다. 전체의 구성도는 [그림

1]과 같다.



[그림 1] 자바 프레임워크의 구성도

CTOC-BR(Class To Optimized Classes - Bytecode tRanslator)은 우선 클래스파일에서 바이트코드를 추출한다. 추출한 바이트코드에서 명령어의 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 상수풀의 정보를 포함하는 코드를 생성한다. CTOC-B의 예를 들면 imul라는 바이트코드의 표현을 mul.i로 나타낼 수 있다.

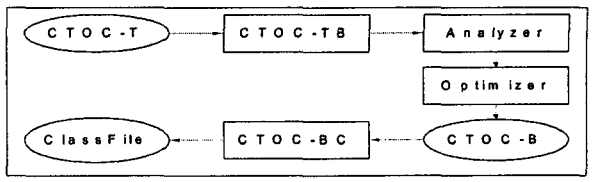
CTOC-TR(Class To Optimized Classes-Three address code tRanslator)는 스택 임시변수를 제거하기 위해서 클래스 파일에서 메소드 단위로 작성한 최대스택크기를 이용한다. 각각의 메소드 단위로 최대 스택 크기만큼 임시변수를 생성하고 타입충돌을 피하기 위해서 타입에 따라 각각의 임시변수를 분리해서 생성한다. 이렇게 생성한 코드는 불필요한 문장을 포함하기 때문에 이러한 문장들을 제거한다.

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 이용하여 최적화된 3주소코드를 생성할 수 있다.

CTOC-RB(Class To Optimized Classes-Return Bytecode)은 CTOC-T를 안전한 스택 기반 코드로 변환하기 위해서 여러 단계를 거쳐서 변환하게 된다. 우선 3주소코드로 변환할 때 AST(Abstract Syntax Tree)를 구성하고, 일반적인 트리 탐색기법인 전위 순회법(preorder traversal)을 이용하여 코드 변환한다. CTOC-B를 oolong의 스택 기반 코드로 변환한 후 oolong을 이용하여 클래스 파일을 생성한다[7].

3. CTOC-RB의 설계

3주소 코드를 효율적인 스택 기반 코드로의 변환기는 CTOC-TB, Analyzer, Optimizer, CTOC-RC등으로 구성되어 있다. CTOC-RB의 전체적인 구성은 [그림 2]와 같다.



[그림 2] CTOC-RB의 구성도

3.1 CTOC-TB

CTOC-TB는 CTOC-T를 CTOC-B로 변환하기 위해서 CTOC-T로 표현된 코드를 메소드 단위로 기본 블록을 생성한다. 생선된 기본 블록 단위로 AST로 구성한다. AST로 구성된 코드를 전위 순회하면서 CTOC-B 코드를 생성한다. 여기에 대한 예제 코드는 [표 1]에 잘 나타나있다.

[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

```
public class Test extends java.lang.Object
{
    public static void main(java.lang.String[])
    {
        word args;
        args := @parameter0: java.lang.String[];
        push 1;
        dup1.i;
        add.i;
        store.i args;
        return;
    }
    public void <init>()
    {
        word this;
        this := @this: Test;
        load.r this;
        invokespecial <java.lang.Object: void <init>()>;
        return;
    }
}
```

(c)CTOC-B

그러나 생성된 CTOC-B는 과다한 store/load문을 가지고 있다. 단순히 CTOC-B로 변환한 코드를 클래스파일로 변환하여 사용한다면 과도한 store/load문으로 인하여 3주소형식을 통한 최적화의 노력이 오히려 실행속도 저하를 가지고 올 수도 있다.

3.2 Analyzer

Analyzer는 흐름분석을 통해 제어 흐름과 데이터 흐름에 대한 정보를 생성하고 생성된 정보는 Optimizer를 위해서 사용된다. 따라서 Analyzer는 제어 흐름을 이용하여 메소드 단위로 기본 블록을 생성한다. 데이터 흐름 분석을 통하여 메소드 단위로 정보를 생성한다.

3.3 Optimizer

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 두 가지로 구분한다.

- store/load
- store/load/load

분류된 패턴들은 Analyzer에서 데이터 흐름 분석에서 메소드 단위로 생성한 정보를 이용하게 된다. 생성된 정보들은 메소드 단위로 변수들의 값의 변화를 체크하는데 사용된다. 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. 다음 패턴인 store/load/load의 패턴의 경우에도 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.

3.4 CTOC-BC

CTOC-BC는 CTOC-B코드를 클래스파일로 변환하기 위해서는 클래스 파일 형식의 이진 화일로 변환을 해야 한다. 이러한 변환 작업을 해주는 oolong이라는 툴을 이용한다. oolong은 바이트코드와 비슷하고 1대1일로 패턴매칭이 가능한 코드이다. 따라서 CTOC-B를 바이트코드와 비슷한 문법을 가진 oolong의 문법에 맞게 변환시켜서 CTOC-B를 클래스 파일로 변환한다.

4. 결론 및 향후연구

자바에 문제점인 실행시간에 문제점에 대한 해결책으로 최적화, JIT컴파일러, 네이티비브 코드로의 변환등 방법등이 나왔으나, 이러한 해결방법들도 결점들을 가지고 있었다. 이러한 결점을 해결하기 위해서 제시한 자바 프레임워크중에서 3주소의 중간언어에서 클래스 파일을 생성할 때 생기는 과다한 store문와 load문을 해결하는 3주소 코드를 효율적인 스택 기반 코드로의 변환기를 설계하였다. 향후 연구과제는 본 논문에서 제시하는 변환기를 구현하고 보다 더 효율적인 코드 생성을 위한 최적화 모듈을 생성하는 것이다.

참고문헌

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

[2] Todd A. Proebsting, Greg Townsend, Patrick Bridges, "Toba: Java For Applications A Way Ahead of Time(WAT) Compiler", COOTS97, pp. 41-53, 1997

[3] 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

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

[5] 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

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

[7] 김영국, 김기태, 조선문, 유원희,"바이트코드 최적화 프레임워크의 설계," 제21회 춘계학술발표대회 제11권 제1호, pp. 297-300, 한국정보처리학회 2004. 05.