

3-주소 코드를 스택-기반 코드로의 변환기

김영국*, 고훈준**, 유원희*

*인하대학교 컴퓨터정보공학과

**경인여자대학 컴퓨터정보기술학부

e-mail : ykkims@spymac.com

Translator for Stack-Based Codes from Three-Address Codes

Young-Kook Kim, Hoon-Joon Kouh, Weon-Hee Yoo

*Department of Computer Science & Information Engineering
Inha University

**School of Computer Information Technology Kyungin
Women's College

요 약

자바의 문제점은 실행속도의 저하이다. 실행속도 저하의 해결 방법으로 네이티브 코드로 변환, JIT 컴파일러, 바이트코드 최적화등의 연구가 되어 왔다. 그중에 바이트코드 최적화 방법을 사용하는 CTOC(Class To Optimized Classes)에서 3-주소 코드를 스택-기반 코드로 코드 확장 기법으로 변환 시 불필요한 store/load 코드가 생성된다.

따라서 본 논문은 불필요한 store/load 코드를 제거하기 위해서 부분 중복 코드 제거 후 불필요한 store/load문을 제거함으로써 불필요한 store/load 코드의 양을 줄이는 변환기를 제안하고, 거기에 대한 간단한 예를 들어 설명한다.

1. 서론

자바는 컴파일 언어에 비하여 실행속도 저하라는 단점을 가진다. 단점을 극복하기 위해 크게 3가지 방향의 연구가 진행 중에 있다.

첫째 클래스 파일을 실행코드로 변환하면 프로그램의 실행속도는 컴파일언어에 비해 저하되지 않지만 플랫폼 독립의 장점을 사용할 수 없다는 단점을 가진다[1].

둘째 JIT 컴파일 방법은 코드를 실행하면 JIT 컴파일러가 일정 시간 동안 바이트코드 중 일부나 전체 코드를 네이티브 코드로 컴파일한다. 따라서 나중에 동일한 메소드를 실행하면 다시 해석하지 않고 시스템 코드를 실행하므로 실행속도가 JVM을 사용하는 것보다 빠르다. 그러나 실행이 반복되는 메소드를 구분하는 알고리즘이나 JIT컴파일 시간이 적당히 조정하는 것이 필요하다는 단점을 가진다[2].

셋째 바이트코드 최적화를 이용한 방법은 기존의 연

구 되었던 최적화 방법들 중 하나 이상의 방법을 조합하여 사용한다. 따라서 바이트코드로 변환되는 모든 언어에 대해서 최적화를 적용할 수 있고, 또한 JVM에서 실행되므로 하드웨어에 독립적으로 사용할 수 있는 장점을 가진다[3].

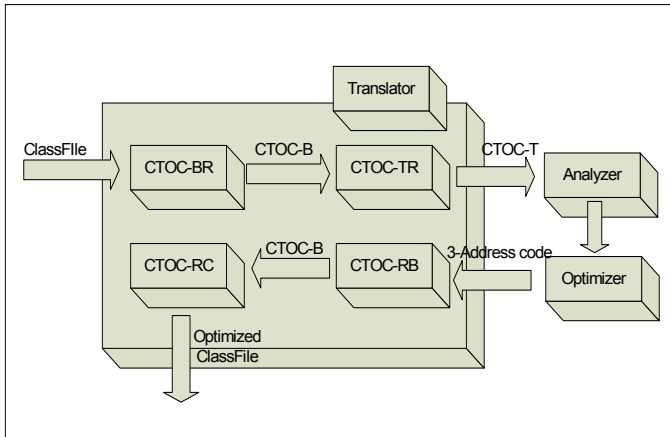
바이트코드 최적화의 장점을 사용할 수 있는 CTOC라는 프레임워크가 존재한다[4]. CTOC에서 3-주소 코드(CTOC-T)를 스택-기반 코드(CTOC-B)로의 변환 시 생기는 불필요한 store/load 코드가 생성된다. 따라서 본 논문에서는 CTOC-T를 CTOC-B로 변환 시 생성되는 불필요한 store/load 코드를 제거하는 방법으로 부분 중복 코드 제거후 store/load 패턴을 이용하여 최적화 시키는 optimizer부분을 추가한 변환기를 제안하고 예제를 들어 설명한다. 그리고 결론과 향후 연구와 방향에 대해서 기술한다.

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

2. 관련연구

2.1 자바 최적화 프레임 워크(CTOC)

자바 최적화 프레임워크는 CTOC-BR(CTOC-Bytecode tRanslator), CTOC-TR(CTOC - Threes addresses codes tRanslator), CTOC-RB(CTOC - Return Bytecode), CTOC-RC(CTOC - Return Classfile)로 이루어진다. 전체의 구성도는 [그림 1]과 같다.



[그림 1] CTOC의 전체 구성도

CTOC는 바이트코드 최적화 기법을 사용하는 프레임워크이다. 따라서 클래스 파일을 입력으로 받아서 3-주소 코드로 변환시킨 후 전통적인 3-주소 형식의 최적화를 사용한다. 3-주소 형식의 최적화된 코드를 클래스 파일로 변환하여 최적화된 바이트코드를 생성하는 프레임워크이다[5].

2.2 CTOC-T에서 CTOC-B로 변환 시 문제점

[그림 2]은 CTOC-T의 간단한 산술 코드 예제이다.

```
public class Exp_Assign2 extends java.lang.Object {
    public static void main(java.lang.String[] {
        java.lang.String[] args;
        int a, b, c;
        args := @parameter0: java.lang.String[];
        a = -10;
        if a >= 0 goto label0;
        b = neg a;
        c = b * b;
        goto label1;
    label0:
        b = 5;
    label1:
        c = b * b;
        return;
    }
    public void <init>() {
        Exp_Assign2 this;
        this := @this: Exp_Assign2;
        specialinvoke this.<java.lang.Object: void <init>()>();
        return;
    }
}
```

[그림 4] CTOC-T의 산술 코드

[그림 2]에서 표현한 예제에서 굵은 글씨로 표현한 코드는 부분 중복 코드를 의미한다. 일반적인 변환

기의 경우에는 부분 중복된 코드에 대해서도 일반 코드와 동일하게 변환한다. 변환된 코드에 대해서 일반적인 최적화 방법으로 store/load 패턴을 이용한다. 일반적인 최적화 방법을 적용해도 불필요한 store/load 코드가 나타난다.

```
public class Exp_Assign2 extends java.lang.Object {
    public static void main(java.lang.String[] {
        word r0, i2;
        r0 := @parameter0: java.lang.String[];
        push -10;
        store.b r0;
        load.b r0;
        ifge label0;
        load.b r0;
        neg.i;
        store.i r0;
        load.i r0;
        load.i r0;
        mul.i;
        store.i i2;
        goto label1;
    label0:
        push 5;
        store.b r0;
    label1:
        load.i r0;
        load.i r0;
        mul.i;
        store.i i2;
        return;
    }
    public void <init>() {
        word r0;
        r0 := @this: Exp_Assign2;
        load.r r0;
        specialinvoke <java.lang.Object: void <init>()>();
        return;
    }
}
```

[그림 3] store/load만 적용한 경우

[그림 3]에서 굵은 글씨로 표현된 코드는 store/load패턴을 이용한 코드 최적화 기법만을 사용할 경우에는 제거될 수 없다.

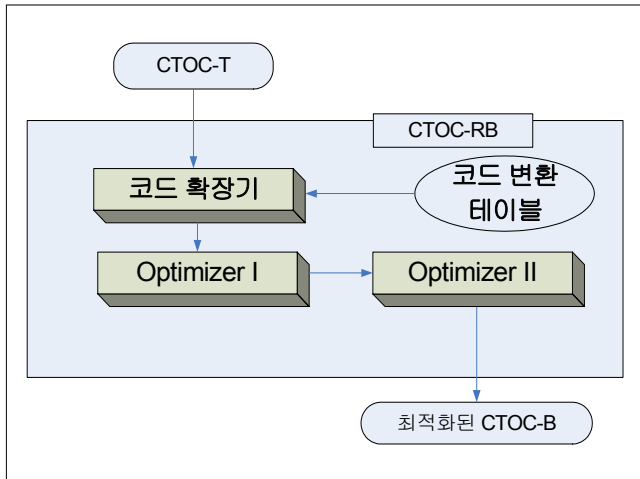
그러나 중복 제거 후에 store/load패턴을 적용할 경우에는 굵은 글씨로 표현된 부분 중 불필요한 코드들을 제거 할 수 있다.

3. CTOC-T를 CTOC-B로의 변환기

본 논문에서 제안하는 CTOC-T를 CTOC-B로 변환하는 변환기는 코드 확장기, 코드 변환 테이블, Optimizer I, Optimizer II로 이루어져 있고, 구성은 [그림 4]와 같다.

코드 확장기는 CTOC-B를 생성하는 핵심 부분으로서 코드 변환 테이블 정보를 참조하여 각 CTOC-T에 대한 코드 확장 기법을 이용한다. 따라서 하나의 CTOC-T에 대해서 동일한 기능을 갖는 CTOC-B코드로 변환된다. 코드 변환 테이블에는 중간 코드 변환 정보에 대한 실질적인 정보를 저장하는 부분으로서 CTOC-T 특성을 고려하여 유사한

기능을 갖는 명령어 그룹으로 구성되어 있다.



[그림 4] CTOC-T를 CTOC-B로의 변환기

Optimizer I은 부분 중복 코드를 제거하는 부분이다. 부분 중복 코드는 공통 부분 수식과 접근 연산 수식을 통틀어서 말한다. 부분 중복 제거는 First-order 식을 사용한다. First-order식은 상수와 변수만을 포함하는 수식을 말한다. 부분 중복 코드 제거의 간단한 [알고리즘 1]은 다음과 같다.

[알고리즘 1]

1. 입력 : CTOC-B코드
2. 출력 : 부분 중복 코드가 제거된 CTOC-B코드
3. 변환 알고리즘
 - ① 프로그램을 전위순서로 정렬시킨 CFG내에 First-order식이 나타났을 경우 worklist에 삽입
 - ② worklist가 비어 있을 경우 정지, 아니면 worklist의 수식(E)에 대해서 선택하거나 제거
 - ③ 수식(E)에 대해서 \emptyset 노드 위치시킴
 - ④ 생성된 수식(E) 대해서 SSA 생성
 - ⑤ 결정된 \emptyset 노드에 대해서 down-safety
 - ⑥ 결정된 \emptyset 노드에 사용 가능한 수식에 대해 코드 삽입과 실행
 - ⑦ 수식 E에 대해서 임시 변수로 저장하고 재로드
 - ⑧ 코드 모션 실행
 - ⑨ ②번으로 돌아감[6]

[그림 2]에서 굵은 글씨로 표현된 부분 중복 코드 ($c=b+b;$)를 제거하기 위해서 부분 중복 코드 제거 알고리즘을 이용한다. [알고리즘 1]을 적용한 내용은 [그림 5]와 같다.

그러나 [그림 5]는 [그림 3]보다 실제로 코드에서 증가된다. 증가된 부분은 임시변수의 생성과 임시변수와 관련된 load/store문 부분이다. 중간에 증가된 부분을 이용하여 Optimizer II에서 최적화를 더 효율적으로 할 수 있다.

```
public class Exp_Assign2 extends java.lang.Object {
    public static void main(java.lang.String[]) {
        word r0, i2;
        r0 := @parameter0: java.lang.String[];
        push -10;
        store.i r0;
        load.i r0;
        ifge label0;
        load.i r0;
        neg.i;
        store.i r0;
        load.i r0;
        load.i r0;
        mul.i;
        store.i i2;
        goto label1;
    label0:
        push 5;
        store.i r0;
        load.i r0;
        load.i r0;
        mul.i;
        store.i i2;
    label1:
        load.i i2;
        store.i i2;
        return;
    }
    public void <init>() {
        word r0;
        r0 := @this: Exp_Assign2;
        load.r r0;
        specialinvoke <java.lang.Object: void <init>()>;
        return;
    }
}
```

[그림 5] Optimizer I을 적용 후

Optimizer II는 생성된 CTOC-B를 메소드 단위로 기본 블록을 생성한다. 생성된 기본블록에 대해서 각 변수별로 정의-사용 고리와 사용-정의 고리에 대한 정보 테이블을 구성하고 변수별로 레이블 표시한다. 이렇게 생성된 정보를 통하여 과도한 store/load코드에 대해서 store/load패턴에 대한 최적화를 실행한다. 적용 알고리즘은 아래에 [알고리즘 2]와 같다.

[알고리즘 2]

1. 입력 : CTOC-B코드
2. 출력 : 불필요한 store/load문이 제거된 CTOC-B코드
3. 변환 알고리즘
 - ① 프로그램의 CFG의 흐름 순서에 따라 기본블록리스트를 worklist에 삽입
 - ② worklist가 비었을 경우 레이블로 표시된 store/load문에 대해서 store/load패턴을 적용, 아니면 worklist의 기본블록을 선택
 - ③ 선택된 기본블록의 사용-정의 고리 정보를 사용하여 제거 가능한 store/load명령어를 같은 레이블로 표시
 - ④ worklist에서 선택된 기본블록 제거
 - ⑤ ②번으로 돌아감

Optimizer II에서 정의된 패턴은 크게 store/load문과 store/load/load문으로 구분한다. 기본 블록 내

에 변수들의 값의 변화를 Optimizer II가 생성한 정보를 이용하여 체크한다. 변수들의 값의 변화가 없을 경우에만 위의 두 패턴이 적용 가능하기 때문이다.

store/load문은 기본블록에서 생성된 변수별 정보 테이블에서 재사용이나 재정의가 없는 변수에만 store위치와 load위치에만 레이블을 이용하여 구분한다. 레이블로 구분된 store/load문의 경우 두 개의 구문을 모두 삭제한다.

store/load/load문의 경우에도 store/load문과 비슷하게 최적화가 적용된다. 레이블로 구분된 store/load/load문이 올 경우에는 마지막에 오는 load문에 위치에 dup문을 적용하고 store/load문에 대한 최적화를 적용하면 store/load문 모두 제거 된다.

```
public class Exp_Assign2 extends java.lang.Object {
    public static void main(java.lang.String[]) {
        word r0, i2;
        r0 := @parameter0: java.lang.String[];
        push -10;
        store.i r0;
        load.i r0;
        ifge label0;
        load.i r0;
        neg.i;
        dup1.i;
        mul.i;
        store.i i2;
        goto label1;
    label0:
        push 5;
        dup1.i;
        mul.i;
        store.i i2;
    label1:
        load.i i2;
        store.i i2;
        return;
    }
    public void <init>() {
        word r0;
        r0 := @this: Exp_Assign2;
        load.r r0;
        specialinvoke <java.lang.Object: void <init>(>>;
        return;
    }
}
```

[그림 6] Optimizwer I 후 Optimizer II을 적용 후

Optimizer II에서 하는 최적화를 Optimizer I후에 적용하면 [그림 6]과 같다. [그림 6]에서 변화된 코드 부분은 굵은 글씨로 표현하였다. 변화된 코드는 이전의 코드보다 효율적인 코드이다. 여기에 대한 내용은 [그림 3]와 [그림 5] 그리고 [그림 6]을 통해서 알 수 있다. 그리고 Optimizer I과 Optimizer II의 순서를 바꾸어서 최적화를 적용한 경우와 Optimizer II만 적용한 경우에는 [그림 3]과 같다.

불필요한 store/load 코드 중에는 일반적으로 사용하는 최적화 방법으로 store/load 패턴만을 이용해서 제거할 경우에도 불필요한 store/load 코드가 남아있

을 수 있음을 [그림 3]에서 보였다. 따라서 [그림 3]에서 제거할 수 없는 코드를 제거하기 위해서 store/load패턴 적용 전에 부분 중복 수식 제거를 사용하면 [그림 6]과 같이 효율적인 코드를 생성한다.

4. 결론 및 향후 연구

자바 실행속도 저하의 해결 방법으로 네이티브 코드로 변환, 바이트코드 최적화, JIT 컴파일러등의 연구가 진행 중이다. 그 중에서 바이트코드 최적화 방법을 사용하는 CTOC중 3-주소 코드에서 스택 기반 코드로 변환하는 변환할 때 생기는 불필요한 코드는 실행속도의 저하의 문제점을 가지고 있었다. 문제점으로 발견된 CTOC-T에서 CTOC-B로 변환할 때 생기는 불필요한 store/load 코드가 생성된다.

불필요한 store/load 코드를 제거하기위해 본 논문에서는 부분 중복 수식 제거 후에 store/load 패턴을 적용함으로써 보다 효과적인 코드를 생성함을 예를 통하여 보였다.

그러나 지금의 변환기는 store/load패턴에 대해서만 뺄셈 최적화를 적용하므로 향후 연구과제는 Optimizer II를 적용한 후에도 남아 있을 수 있는 불필요한 코드에 대한 뺄셈 최적화 패턴을 정의하고 적용하는 것이다.

참고문헌

- [1] Ronald Veldema, "JCC, a native Java compiler", Technical report, 1998
- [2] Frank Yellin, "The JIT Compiler API", http://java.sun.com/docs/jit_interface.html, 1996
- [3] Taiana Shpeisman, Mustafa Tikir, "Generating Efficient Stack Code for Java", Technical report, University of Maryland, 1999
- [4] 김영국, 김기태, 조선문, 유원희, "바이트코드 최적화 프레임워크의 설계," 제21회 춘계학술발표대회 제11권 제1호, pp. 297-300, 한국정보처리학회 2004. 05.
- [5] 김영국, 김기태, 조선문, 유원희, "CTOC에서 중간 코드에서 효율적인 바이트코드로의 변환기 설계," 한국정보처리학회 2004년 추계 학술대회, pp. 445-448, 2004.
- [6] Nathaniel John Nystrom, "BYTECODE-LEVEL ANALYSIS AND OPTIMIZATION OF CLASSES", Master's Thesis, Purdue University, August 1998