Journal of Korea Multimedia Society Vol. 10, No. 5, May 2007(pp. 687-697)

CTOC에서 코드 최적화 수행

김기태[†], 유원희[††]

요 약

자바 바이트코드는 많은 장점을 가지고 있지만, 수행속도가 느리고 프로그램 분석이나 최적화에 적절한 표현이 아니라는 단점이 존재한다. 따라서 네트워크와 같은 실행 환경에서 효율적으로 수행되기 위해서는 최적화된 코드로 변환이 필요하다. 따라서 우리는 최적화된 코드로의 변환을 위해 CTOC를 구현하였다. 최적화 과정에서 CTOC는 정적으로 값과 타입을 결정하기 위해 변수를 배정에 따라 분리하는 SSA Form을 사용하였다. 하지만 SSA Form 변환 과정에서 ∅-함수의 추가에 의해 오히려 노드의 수가 증가되는 문제점이 발생하였다. 이를 해결하기 위해 본 논문에서는 SSA Form에서 복사 전파와 죽은 코드 제거 최적화를 수행한다. 또한 기존의 SSA Form은 표현식보다는 주로 변수에 관련된 것이라는 단점이 존재한다. 따라서 본 논문에서는 SSA Form 형태의 표현식에 대해 복사 전파와 죽은 코드 제거와 같은 최적화를 적용한 후 다시 중복된 표현식을 제거하는 과정을 추가로 수행한다.

Processing of Code Optimization in CTOC

Ki-Tae Kim[†], Weon-Hee Yoo[††]

ABSTRACT

Although the Java bytecode has numerous advantages, there are also shortcomings such as slow execution speed and difficulty in analysis. Therefore, in order for the Java class file to be effectively executed under the execution environment such as the network, it is necessary to convert it into optimized code. We implemented CTOC for transforming to optimized code. In Optimization, SSA Form that distinguish variable by assignment is used to determine value and type statically. Copy propagation, dead code elimination optimization is applied to SSA Form. However, existing SSA Form is related to variable than expression. Therefore, in this paper, to performing optimization to SSA Form expression, after performing copy propagation and dead code elimination, in addition to that, partial redundant expression elimination is performed

Key words: CTOC(CTOC), Optimization(최적화), Copy Propagation(복사 전파), Dead Code Elimination (죽은 코드 제거), Partial Redundant Elimination(부분 중복 제거)

1. 서 론

바이트코드는 스택 기반 코드이기 때문에 수행 속도가 느리고, 바이너리 형태의 코드이기 때문에 프로그램 분석이나 최적화에 적절한 표현이 아니라는 단점이 존재한다[1,2]. 따라서 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화 코드로 변환 되어야 한다[3].

최적화 코드로 변환하기 위해 CTOC(Classes To Optimized Classes)를 구현하였다[4-7]. CTOC는 현재 개발 중인 코드 최적화 프레임워크로 McGill 대학의 SOOT 프로젝트와 Purdue 대학의 Bloat 프로젝

※ 교신저자(Corresponding Author) : 김기태, 주소 : 인천 남구 용현동 253 인하대학교(403-751), 전화 : 032)860-7444, FAX : 032)867-7444, E-mail : kkt@inha.ac.kr
접수일 : 2006년 11월 14일, 완료일 : 2007년 3월 12일

[†] 준회원, 인하대학교 컴퓨터 · 정보공학과
[††] 인하대학교 컴퓨터 · 정보공학과
(E-mail : whyoo@inha.ac.kr)

트에 기반을 두고 있다[8,9].

CTOC에서는 바이트코드의 분석과 최적화를 위해 가장 먼저 제어 흐름 분석을 수행하였다[4]. 그 결과로 확장된 CFG인 eCFG를 생성하였고, 이후 변수를 정적으로 다루기 위해 변수를 정의(def)와 사용(use)에 따라 분리했다. 왜냐하면 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문이다. 전통적인 컴파일러에서는 정의와 사용에 대한 정보를 정의-사용 고리(du-chain)로 유지한다[10-12]. 하지만 CTOC에서는 정의-사용 고리 대신 SSA Form을 사용하였다[5]. SSA Form(Static Single Assignment Form)은 이름과 같이 각 변수에 대해 유일한 정의를 갖는 특징을 가진다. 또한 이전 연구를 통해 SSA Form에서 정적 타입 설정에 대한 연구가 수행되었다[6]. 하지만 바이트코드를 SSA Form으로 변환하는 과정에서 병합 지점(join node)에 Ø-함수의 추가에 의해 노드의 수가 증가되는 문제점이 발생하였다.

따라서 본 논문에서는 증가된 노드의 수를 줄이기 위해 우선 SSA Form 형태에서 적용 가능한 최적화 기법인 복사 전파와 죽은 코드 제거를 수행하여 기본적인 최적화를 수행한다. 기본 최적화를 통해서도 기존의 코드보다는 최적화된 코드를 획득할 수 있다. 하지만 SSA Form은 변수에 대한 최적화를 수행하기에는 적절한 형태이지만 표현식에 대한 최적화를 적용하기에는 적절하지 않다는 단점이 여전히 존재한다. 따라서 SSA Form으로 변경된 표현식에 대해 추가 최적화를 적용하기 위해 부분 중복 표현식 제거를 수행한다.

본 논문은 CTOC에서 SSA Form의 코드에 대해 우선 복사 전파와 죽은 코드 제거를 수행한 후 더욱 최적화된 코드를 생성하기 위해 표현식에 대한 부분 중복 제거를 다시 수행한다. 또한 부분 중복 제거 이후 다시 변화된 구조에 대해 다시 복사 전파와 죽은 코드 제거를 수행하여 좀 더 적은 크기의 최적화 코드가 생성되는가를 실험을 통해 확인한다.

본 논문의 구성은 다음과 같다. 2장에서는 최적화의 의미와 관련 연구에 대해 살펴보고, 3장에서는 CTOC에서 복사 전파, 죽은 코드 제거하기, 부분 중복 제거하기와 같은 최적화 수행 과정을 보인다. 4장에서는 각각의 최적화에 대해 비교 실험을 수행하고, 마지막으로 5장에서는 결론과 향후 계획을 논한다.

2. 관련연구

2.1 복사전파

복사 전파란 x = y와 같은 배정문이 주어진 경우 x 또는 y의 값에 변화가 없는 동안 x가 사용되는 부분에 x대신 y를 사용하는 것이다[11]. 복사 전파에는 데이터흐름 분석을 이용하는 것이 일반적인 방법이다. Aho는 x = y에 대해서 정의-사용 고리 정보와 데이터 흐름 분석을 통해서 복사 전파를 수행하였다[10]. Muchnick은 복사 전파를 지역과 전역으로 나눴다[12]. 지역적인 복사 전파는 분할된 기본 블록 내에서 수행되고, 전역 복사 전파는 전체 흐름 그래프를 통해 동작하는 것이다. 특히 ACI(available copy instruction) 테이블이라 불리는 복사 가능한 명령어에 대한 테이블을 유지하면서 복사 전파를 수행하였다. 또한 전통적인 복사 전파에서는 데이터흐름 분석을 적용하기 위해 일반적으로 비트 벡터를 많이 이용해서 표현하였다. 하지만 CTOC에서는 복사 전파를 수행하기위해 비트 벡터 대신 트리 구조와 사용과 관련된 리스트를 이용한다. CTOC에서 복사 전파를 수행하는 최적화 과정은 트리의 각 노드를 방문하면서 수행된다.

2.2 죽은 코드 제거

변수가 프로그램의 특정 지점 이후에 계속 사용될 수 있다면 그 변수는 살아있는 것이고 그렇지 않은 경우에 그 변수는 죽은 것이다. 또한 결코 사용되지 않을 값을 계산하는 문장도 죽은 코드에 해당한다[10]. 일반적으로 최적화 과정에서 의도적으로 죽은 코드를 생성하지는 않지만, 많은 프로그램은 보통 최적화 이전에 죽은 코드를 포함하고 있다. 또한 출력 문장에 영향을 주지 않거나, 연산 강도 경감 또는 복사 전파와 같은 최적화 수행과정을 통해 해당 변수가 다른 변수로 대체된 경우라면 죽은 코드가 발생할 수 있게 된다.

전통적인 죽은 코드 제거 기법은 계산되는 모든 명령어를 마크하면서 시작된다. 만약 프로시저에 의해 출력되거나 명확하게 반환되는 경우이거나, 프로시저의 외부로부터 접근될 수 있는 저장 위치에 영향을 주는 경우라면, 그 값은 살아있는 것이다. 따라서 죽은 코드 제거 알고리즘은 반복적으로 수행되면서

살아있는 값을 이용하는 명령어를 마크한다. 알고리즘 수행 후 마크되지 않은 명령어는 죽은 것이기 때문에 제거해도 프로그램 수행에 영향을 주지 않게 된다.

일반적으로 죽은 코드의 확인은 데이터흐름 분석처럼 공식화할 수 있고 작업리스트와 정의-사용 고리를 이용하여 수행된다. 하지만 CTOC에서는 정의-사용 고리를 사용하는 대신 SSA Form을 사용하고, 데이터흐름 분석 대신 트리 구조를 사용하여 변수에 대한 정보를 유지한다. 또한 복사 전파가 죽은 코드 제거 이전에 수행되면 복사 전파를 통해 복사 문장을 죽은 코드로 바꾸어줄 수 있기 때문에 더욱 효율적으로 수행할 수 있게 된다. 따라서 복사 전파를 수행한 후 죽은 코드 제거 과정을 수행한다.

2.3 부분 중복 제거

프로그램 수행 과정에서 동일한 표현식을 두 번 이상 평가하는 것은 바람직하지 않다. 만약 프로그램에서 a + b 표현식이 아무런 변화 없이 다시 사용된다면 뒤에 평가되는 식은 중복된 것이다[12]. 처음 평가된 a + b의 계산 결과를 임시변수 t를 사용하여 저장한 후, 다음에 나타나는 a + b 대신 이 임시변수 t로 대체하면 중복된 계산을 피할 수 있게 된다. 즉, 중복 표현식 제거는 제어 흐름 경로에서 불필요하게 발생하는 표현식의 중복된 계산을 제거하는 것이다. 하지만 프로그램 내에는 부분 중복이 발생하는 경우가 있다. 부분 중복 제거가 일반적인 표현식 중복 제거와 다른 점은 표현식의 병합 지점에서 발생하는 것이다. 병합 지점에 임시변수를 사용하기 위해서는 병합 지점의 모든 선행 노드에 중복된 표현식이 존재해야 하는데, 부분 중복 표현식은 일부 선행 노드에만 중복된 표현식이 존재하는 경우를 의미한다.

부분 중복 제거는 Morel과 Renvoise에 의해 처음으로 제안된 강력한 최적화 기술이다[13]. 부분 중복 제거는 기존의 전역 공통 부분식 제거(global common subexpression elimination)과 루프 불변 코드 이동(loop invariant code motion)이 결합된 기술이다. Knoop et al.은 지연 코드 이동(lazy code motion)이라고 불리는 배치 전략으로 불필요한 코드의 이동을 피하여 Morel과 Renvoise의 결과를 향상시켰다[14]. Muchnick은 부분 중복 제거의 문제를 소개하고, 이 문제를 해결하는 고전적인 방법을 소개하였다[12]. 부분 중복 제거에 대한 대부분의 접근은 해결하고자 하는 문제를 비트 벡터와 데이터 흐름 방정식 형태로 변형하고 반복적인 형태의 해결책을 제시하였다. Chow et al.는 SSAPRE라 불리는 새로운 접근법에 대해 소개하였다[15].

SSA Form은 기본적으로 프로그램 변수에 관한 문제를 해결하는 것에 제한된다. 왜냐하면 표현식을 위한 정의-사용의 개념이 변수를 위한 경우보다 덜 명확하기 때문이다. 따라서 SSA Form은 표현식에 대한 문제를 해결하기 위해 쉽게 적용될 수 없었다. 하지만 CTOC에서는 비트 벡터에 기반한 전통적인 데이터 흐름 분석을 프로그램 표현을 위한 SSA Form에 잘 적용되도록 변환한다. 그리고 일반적인 표현식 중복과 달리 부분 중복은 병합 지점에서 일부 선행자에서만 중복이 발생하기 때문에 완전한 중복이 될 수 있도록 표현식을 복사하고, 병합 지점에 새로운 P-문장을 추가하는 방법을 사용한다.

3. CTOC의 최적화 수행

논문에서는 CTOC에서 최적화 수행 과정을 서술하기 위해 그림 1과 같은 간단한 소스를 사용한다.

그림 1 (a)의 소스를 SSA Form으로 변환하기 위해서는 우선 그림 1의 (b)와 같은 바이트코드를 CTOC의 입력으로 사용한다. 실제 CTOC의 입력은 (b)와 같이 역어셈블된 형태가 아닌 바이너리 파일인 .class 파일이다. 하지만 여기서는 이해를 돕기 위해 역어셈블된 코드로 표현하였다.

```
public int f(boolean b, int i, int j){
    int k, l;
    if(b)
        k = i + j;
    else
        k = 3;
    l = (i + j) * k;
    return l;
}
```

```
public int f(boolean,int,int);
  Code:
    0:  iload_1
    1:  ifeq        12
    4:  iload_2
    5:  iload_3
    6:  iadd
    7:  istore      4
    9:  goto        15
   12:  iconst_3
   13:  istore      4
   15:  iload_2
   16:  iload_3
   17:  iadd
   18:  iload       4
   20:  imul
   21:  istore      5
   23:  iload       5
   25:  ireturn
```

그림 1. (a) 소스 (b) 바이트코드

SSA Form으로 변환한 과정 후에 생성된 eCFG는 그림 2와 같다.

그림 2(a)는 각 기본 블록에 대해 변환된 코드를 보여준다. 또한 이 과정에서 (b)와 같은 형태의 eCFG가 생성된다. 각 블록에 존재하는 문장들은 트리 형태로 존재한다. 이 때 사용되는 트리를 표현 트리(*expression tree*)라 하는데 기본 블록 내부에서 각 명령어를 트리 형태의 문장으로 표현하기 위해 사용된다. 표현 트리는 커다랗게 추상 클래스인 <*Expression*> 클래스로부터 파생된 표현식과 역시 추상 클래스인 <*Statement*> 클래스로부터 파생된 문장으로 나눠진다. 그림 3은 표현 트리를 구성하는 BNF의 일부를 나타낸다.

그림 3의 BNF를 통해 <*Expression*>과 <*Statement*>의 파생 클래스를 생성한다. 이 두 종류의 클래스의 차이는 <*Expression*>으로부터 파생된 클래스들은 값을 유지할 수 있는 *val* 필드를 가진다는 것이다. 또한 각 <*Expression*>은 타입을 표현하기 위해 *type* 필드도 추가로 가진다.

```
Statement  › ExprStmt | InitStmt | JumpStmt | LabelStmt | PHIStmt
LabelStmt  › Label
InitStmt  › INIT  LocalExpression | }
ExpressionStatement  › evaluation  Expression
JumpStmt  › GotoStmt | IfStmt | ReturnExprStmt
IfStmt  › IfZeroStmt
GotoStmt  › goto  Block
IfZeroStmt  › if0(Expression(==|!=|>|>=|<|<=)(null |0)) then
          Block  else  Block
ReturnExprStmt  › return Expression
PHIStmt  › PHIJoinStmt
PHIJoinStmt  › Statement:=PHI(Label=Expression,Label=Expression)
Block  › <block  Label>
Label  › label_ Num
Expression  › ConstantExpression | DefExpr | StoreExpression | BinExpr
DefExpr  › MemExpression
BinExpr  › ArithExpression
StoreExpression  › (  MemExpression  :=  Expression )
MemExpression  › VarExpr
VarExpr  › LocalExpression | StackExpression
ArithExpression  › Expression Op Expression
LocalExpression  › (Stack | Local ) Type  Num (_undef | _Num)
ConstantExpression  › ' ID '  | Num
```

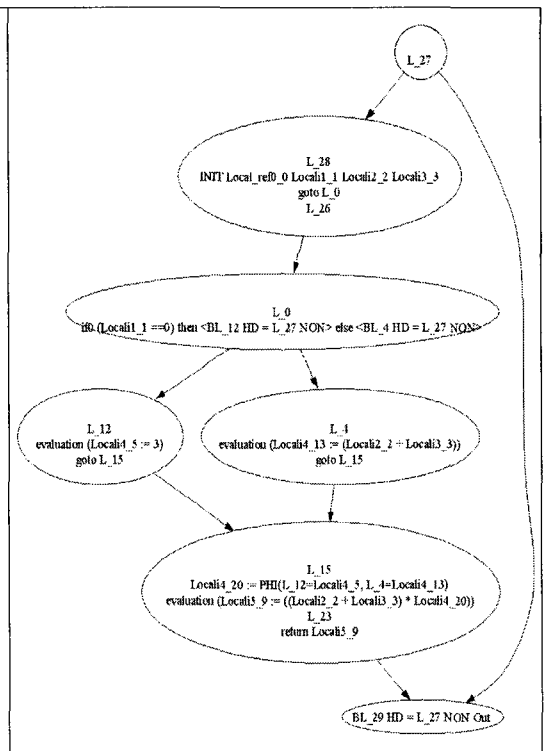
그림 3. BNF의 일부

3.1 CTOC에서의 복사 전파

CTOC에서 복사 전파는 트리의 각 노드를 전위

```
<BL_27 ENTRY>
    L_27

<BL_28 INIT>
    L_28
        INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3
        goto L_0

<BL_0>
    L_0
        if0 (Locali1_1 ==0) then <BL_12> else <BL_4>

<BL_4>
    L_4
        evaluation (Locali4_13 := (Locali2_2 + Locali3_3))
        goto L_15

<BL_12>
    L_12
        evaluation (Locali4_5 := 3)
        goto L_15

<BL_15>
    L_15
        Locali4_20 := PHI(L_12=Locali4_5, L_4=Locali4_13)
        evaluation (Locali5_9 := ((Locali2_2 + Locali3_3) * Locali4_20))
    L_23
        return Locali5_9

<BL_29 EXIT>
    L_29
```

(a)



(b)

그림 2. (a) SSA Form이 적용된 eCFG 코드 (b) eCFG

순서(pre-order)로 방문하면서 수행한다. 우선 CTOC에서 복사 전파가 발생할 수 있는 구조를 보면 그림 3의 BNF에서 <*StoreExpression*>이 이에 해당한다. <*StoreExpression*>의 트리 구조는 그림 4와 같다.

그림 4의 <*StoreExpression*>의 구조를 보면 := 를 기준으로 왼쪽에 <*MemExpression*>이 있고 오른쪽에 <*Expression*>의 파생 클래스가 올 수 있는 구조를 가진다. <*Mem Expression*>이란 메모리를 의미하는 것으로 배정문에서 좌측 값(*l-value*)을 의미한다. 본 논문에서는 좌측 값을 *lhs*로 나타내었다. 그림 3에서는 *lhs*로 <*VarExpr*>을 사용한다. <*VarExpr*>은 일반적인 변수를 의미한다. 오른쪽은 배정문에서 우측 값(*r-value*)으로 *rhs*로 나타낸다. *rhs*로 <*Expression*>이 올 수 있는데 대표적인 <*Expression*>의 파생 클래스로 <*ConstantExpression*>, <*DefExpr*>, <*StoreExpression*>, 그리고 <*ArithExpression*> 등이 올 수 있다. 특히 복사 전파에서는 <*StoreExpression*>의 우측 값에 다시 <*StoreExpression*>이 나타나는 경우를 고려해야 한다. 왜냐하면 이러한 경우가 복사 전파가 발생할 수 있는 경우이기 때문이다.

예를 들면 <*StoreExpression*>으로 *(Locali2_1 := 1)*와 *(Locali1_2 := (Locali2_1 := 1))*의 경우가 나타날 수 있다. 두 경우 중에 첫 번째는 단순히 배정이 발생하지만 두 번째 경우는 복사 전파를 적용할 수 있는 경우에 해당한다. 이를 위해 알고리즘 1을 사용한다.

알고리즘 1에서 <*StoreExpression*>인 *(Locali1_2 := (Locali2_1 := 1))*의 경우 *lhs*는 *Locali1_2* 이고 *rhs*는 *(Locali2_1 := 1)*인 상태이다. 즉 현재의 *rhs*가 또 다시 <*Store Expression*>인 상태이다. 따라서 알고리즘 1이 다시 적용 가능하다. 현재 *rhs*를 *store*로 복사하고, *store.lhs()*로 *rhs*의 *lhs*를 새롭게 구해 *rhsLHS*에 저장한다. 그리고 *store.rhs()*로 *rhs*의 *rhs*를 *rhsRHS*에 저장한다. 이렇게 수행한 후 *rhsLHS*

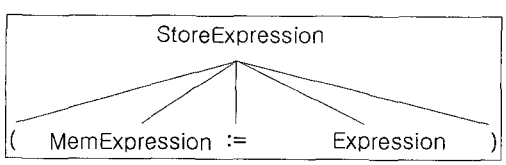


그림 4. 〈StoreExpression〉 구조

알고리즘 1. 복사 전파 알고리즘

```
Input : expr ∈ StoreExpression
Output : changed ∈ Boolean
procedure copyProp(expr)
begin
    changed ← false
    if (rhs is instance of StoreExpression)
        store ← rhs
        rhsLHS ← store.lhs()
        rhsRHS = store.rhs()
        if (rhsLHS is instance of LocalExpressjon)
            copy ← lhs.clone()
            copy.setDef(lhs)
            if (propExpr(expr.block(), rhsLHS, copy))
                changed ← true
                expr.visit(Replace(rhs, rhsRHS))
                rhsLHS.cleanup()
                rhs.cleanupOnly()
            fi
            copy.cleanup()
        fi
    fi
    return change
end
```

에는 *Locali2_1*이, *rhsRHS*에는 1이 저장된 상태이다. 다음은 *rhsLHS*가 <*LocalExpression*>인가를 확인한다. 이때 이전에 생성한 *lhs*인 *Locali1_2*에 대한 복사본을 *copy*로 생성한다. 이 *copy*는 기존의 *lhs*와 동일한 성질을 갖는 새로운 노드이다. 또한 *propExpr(block, rhs, lhs)*메소드를 통해서 복사 전파가 가능한가를 확인한다. *propExpr (block, rhs, lhs)*메소드는 해당 블록에서 *rhs*를 *lhs*로 전파하는 동작이 수행 가능한가를 확인하게 된다. 즉, *rhs*에 해당하는 *Locali2_1*을 *Locali1_2*로 대체할 수 있는가를 확인한다. 실제 대체하는 동작은 *Replace* 클래스를 통해서 이루어진다. 이 클래스는 첫 번째 표현식을 두 번째 표현식으로 대체하는 동작을 수행한다. 즉, 이 클래스를 통해 실제로 *Locali2_1*을 *Locali1_2*로 대체하게 된다. 대체를 수행한 후 *rhsLHS*인 *Locali2_1*은 더 이상 필요 없는 노드이기 때문에 제거 가능하게 된다. *rhs*인 *(Locali2_1 := 1)*도 제거해야 한다. 왜냐하면 *Locali2_1*이 제거되었기 때문이다.

3.2 CTOC에서의 죽은 코드 제거

CTOC에서 죽은 코드 제거 과정을 수행하기 위해서는 우선 죽은 노드를 확인할 수 있는 *live* 필드를

모든 노드에 새롭게 추가한다. 이후 eCFG의 모든 노드를 전위 순서로 방문하면서 *live* 필드에 *DEAD*로 초기 값을 설정한다. *DEAD*는 블리언 값으로 *fasle*를 의미한다. 본 논문에서는 죽은 노드라는 의미를 명확하게 하기 위해 *DEAD*를 사용하고 *true* 대신 *LIVE*를 사용한다.

모든 노드를 *DEAD*로 설정한 후 죽은 코드 제거 과정을 수행하기 전에 특정 조건에 맞는 문장과 표현식들은 미리 *LIVE*로 다시 설정해야 한다. 왜냐하면 이들 문장과 표현식은 프로그램에서 항상 살아있는 경우이기 때문이다. 미리 *LIVE*로 설정되어야하는 경우는 다음과 같다. 첫째, 문장이 프로그램의 출력에 영향을 주는 경우, 둘째, 문장이 배정문이고 배정문중 *l-value*에 해당하는 *lhs*가 살아있는 문장에서 사용되는 경우, 셋째, 문장이 조건문이고 제어에 의해 하나 또는 그 이상의 문장이 실행되는 경우이다. 그림 3의 BNF에서 초기화 문장을 의미하는 <*Init-Stmt*>는 항상 *LIVE*하다고 가정한다. 그리고 배정문을 의미하는 <*Store Expression*>은 *lhs*가 <*LocalExpression*>인 경우에 *LIVE*하다고 설정한다. 또한 <*IfStmt*>, <*GotoStmt*>, 그리고 <*Return-ExprStmt*>와 같이 분기되는 문장 역시 *LIVE*하다고 미리 설정해야 한다.

트리로부터 죽은 노드를 제거하기 위해서는 다시 eCFG의 각 문장을 전위 순서로 읽어 들인다. 그 후 읽어드린 노드의 *live* 필드의 값을 확인한다. *live* 필드의 값이 *DEAD*라면 해당 노드를 트리로부터 제거해야 하고, 그 외의 경우는 *LIVE*인 상태가 된다. 하지만 *DEAD*인 노드 중에 라벨을 의미하는 <*LabelStmt*>와 제어의 이동을 의미하는 <*Jump-Stmt*>는 프로그램 내에서 분기에 관련된 정보를 나타내기 때문에 제거하지 않는다.

3.3 CTOC에서의 부분 중복 제거

CTOC에 의해 생성된 SSA Form인 그림 2를 보면 <BL_4>와 <BL_15>에 표현식 *Locali2_2 + Locali3_3*이 중복된 것을 확인할 수 있다. 그러나 현재 표현식이 <BL_4>와 <BL_15>에만 존재하고 <BL_12>에는 존재하지 않기 때문에 전체 중복이 아니라 부분적으로 중복된 상태이다. 기존의 경우에서는 이러한 경우 중복 표현식 제거가 불가능 했지만 분석을 통해 부분 중복된 표현식을 복사하여 전체 중복으로 변환하면 이 경우도 제거가 가능하게 된다. 즉, 부분 중복된 표현식을 전체 중복으로 만들기 위해 본 논문에서는 표현식을 임시 변수로 대체하고 부분 중복된 표현식을 복제하여 병합지점의 선행자에 해당 표현식을 이동시켜 완전한 중복 형태로 변형한다. 그리고 SSA Form에서 변수에 사용된 것과 유사하게 중복된 표현식을 의미하는 임시 변수들을 병합 지점에서 선택하도록 하였다.

SSA Form에서 부분 중복 제거를 수행하기 위해 우선 문장 내에서 부분 중복을 가질 수 있는 표현식을 먼저 찾아야 한다. 부분 중복을 가질 수 있는 a + b와 같은 표현식을 <*BinExpr*>라 정의하였다. 예를 들면 <*BinExpr*>은 그림 3의 BNF에 있는 <*Arith Expression*> 형태를 의미한다. <*BinExpr*>이 될 수 있는 <*ArithExpression*>의 구조는 그림 5와 같다.

<*ArithExpression*>은 그림 5와 같이 하나의 <*Op*>와 두개의 <*Expression*>을 가지는데 각 <*Expression*>은 트리에서 마지막 노드여야 한다. 마지막 노드가 되기 위한 조건은 각 <*Expression*>이 그림 3에서 <*LocalExpression*>이거나 <*Constant-Expression*>이면 된다. 즉, 지역 변수이거나 상수 값이 나타나면 마지막 노드라는 의미이다.

제거 가능한 표현식을 찾기 위해 다음에 수행할 일은 동일한 표현식이 어느 곳에 나타나는가를 찾은 후 작업리스트에 추가하는 것이다. 우선 그림 2의 SSA Form 형태의 eCFG를 통해 해당 블록과 각 문장을 방문하여 문장에 <*ArithExpression*>이 존재하는가를 확인한다. 각 블록은 전위 순서로 방문한다. 그림 6는 <BL_4>의 <*ExpressionStatement*> 문장인 *evaluation (Locali4_13 : = Locali2_2 + Locali3_3)*을 방문하는 경우를 나타낸다.

우선 각 문장에서 표현식을 만나게 되면 해당 표현식이 <*BinExpr*>인가를 확인한다. 그림 6에서 처음 방문한 문장인 <*ExpressionStatment*>는 <*State-ment*>를 상속 받아 생성된 문장이기 때문에

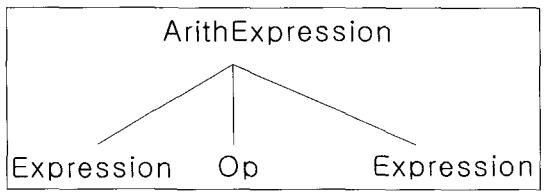


그림 5. 〈*ArithExpression*〉 구조

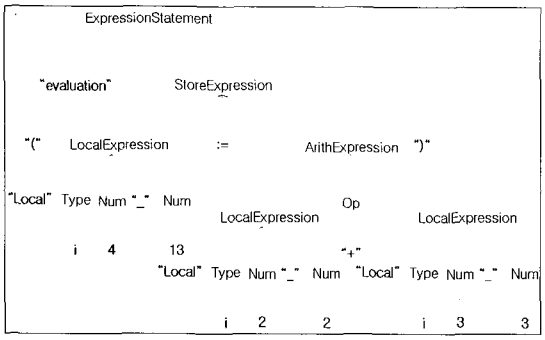


그림 6. 〈*ExpressionStatement*〉 구조

<*BinExpr*>를 체크하지 않는다. 다시 <*Expression-Statement*>의 자식인 <*StoreStatement*>를 방문하면, 이 경우에도 <*Statement*>클래스를 상속 받아 생성된 경우이기 때문에 <*BinExpr*>를 고려하지 않는다. <*StoreStatement*>는 그림 6과 같이 :=의 좌측에 <*LocalExpression*>과 우측에 <*ArithExpression*>으로 나누어진다. 좌측은 <*Local Expression*> 형태만이 올 수 있는 위치이기 때문에 역시 <*BinExpr*>를 고려하지 않는다. <*Expression*>의 경우에는 <*Expression*>으로부터 상속 받은 표현식들이 존재할 수 있는 위치이기 때문에 반드시 <*BinExpr*>에 대한 체크를 수행해야 한다. 그림 6의 경우는 <*Expression*>에 <*ArithExpression*>이 존재하는 경우이다. 하지만 모든 <*Arith Expression*>이 <*BinExpr*>는 아니다. 예를 들면 a + (b + c) 와 같이 중복된 경우도 존재하기 때문이다. 이러한 경우가 존재하는가를 확인하기 위해 현재 방문한 노드가 <*ArithExpression*>인 경우 <*ArithExpression*>의 왼쪽과 오른쪽이 각각의 마지막 노드인가를 확인한다. 만약 중복된 경우 없이 단순히 해당 노드가 <*LocalExpression*>이거나 <*ConstantExpression*>인 경우라면 트리에서 마지막 노드에 해당하기 때문에 해당 <*ArithExpression*>을 <*BinExpr*>라고 할 수 있다. 하지만 왼쪽이나 오른쪽의 표현식이 또 다시 <*ArithExpression*>을 갖는 경우라면 현재의 표현식은 <*BinExpr*>가 아닌 경우가 된다. 이렇게 생성된 <*BinExpr*>인 표현식은 다른 블록에서 중복된 표현식이 존재하는가를 확인하기 위해 작업리스트에 추가하게 된다. 작업리스트는 앞의 경우와 같이 모든 트리의 노드를 방문하면서 <*ArithExpression*>을 찾고 해당 표현식이 <*BinExpr*>의 조건을 만족한다면 작업리스트에 계속 추가시킨다.

<*BinExpr*>가 존재하는 블록에 대한 정보를 획득한 후 부분 중복 표현식 제거를 위해 P-문장을 eCFG에 삽입해야 한다. P-문장은 SSA Form의 Ø-함수와는 조금 다르다. SSA의 Ø-함수는 변수에 관한 것이지만 P-문장은 표현식과 관련된 것이기 때문이다.

P-문장은 프로그램에서 표현식의 다른 값이 병합되는 지점에 도달할 때 요구된다. 다음 두 가지 경우엔 반드시 P-문장이 삽입되어야 한다. 첫 번째, P-문장은 표현식이 존재하는 블록에 대한 반복적 지배경계(*iterated dominance frontier*)가 되는 블록에 삽입된다. 두 번째, P-문장은 반드시 표현식에 있는 변수들 중 하나를 위해 Ø-함수를 포함하는 블록에 삽입된다. 즉, 기존 eCFG에서 Ø-함수가 존재하는 블록에 P-문장을 삽입하게 된다. P-문장은 변수가 재정의 되고, 따라서 그 변수를 사용하는 표현식이 새로운 값을 가질 수 있다는 것을 나타낸다. 그림 2의 SSA Form의 eCFG에서 위의 2가지 조건을 적용하여 P-문장이 삽입될 수 있는 위치는 <BL_15>와 <BL_29>이다. 일반적으로 마지막 블록은 표현식이 존재하는 블록에 대한 반복적인 지배자 경계인 경우라도 P-문장을 추가하지 않는다. 왜냐하면 마지막 블록 다음에는 더 이상의 표현식이 존재하지 않기 때문이다. 따라서 불필요한 <BL_29>의 P-문장은 추가하지 않는다.

모든 준비가 끝난 후에는 코드를 이동하여 부분 중복인 표현식을 완전한 중복으로 만들어야 한다. <BL_15>에 P-문장이 존재하는 경우, 현재 블록의 선행자를 방문하여 선행자 블록에 기존의 피연산자 정보가 존재하는가를 확인하였다. <BL_15>의 P-문장이 *PHI(L_4=1, L_12=UDef)*인 상태이기 때문에 선행 블록 <BL_12>에 피연산자에 대한 정보가 존재하지 않은 상태이다. *UDef*는 정의가 존재하지 않는다는 의미이다. 부분 중복이 발생한 것을 완전한 중복이 발생하도록 하기 위해서 표현식이 존재하지 않는 선행자 블록에 새로운 문장을 추가하고 이 문장을 P-문장의 피연산자로 사용하게 된다. 이를 위해서 *CodeMove()* 메소드를 호출한다. 이 메소드는 피연산자를 P-문장을 의미하는 <*PHIStmt*>의 피연산자로 생성하는 동작을 수행한다. 이 과정에서 임시로 생성된 표현식에 새로운 값 번호(*VN : Value*

Number)를 설정한다[7]. 이 *VN*은 P-문장에서 사용될 임시변수 t와 같은 *VN*을 갖게 된다. 다음은 이 값을 임시변수로 대체하기 위해 <*StoreExpression*> 객체를 새롭게 생성한다. <BL_12>에 이동된 코드 형태는 *(Locali6_31 := (Locali2_2 +Locali3_3)) [VN] : 36* 이다. *[VN] : 36*은 추가된 문장에 설정된 새로운 값 번호를 함께 나타낸 것이다. 표현식을 임시변수에 배정하고 모든 노드에 같은 *VN*을 설정하게 된다. 생성된 문장은 표현식이 존재하지 않는 선행자의 마지막 <*JumpStatement*> 앞에 위치하게 된다. 즉 그림 7(b)와 같이 <BL_12>의 goto 문장 앞에 위치하게 된다.

그림 7의 (a)의 *evaluation (Locali4_5 := 3)* 문장이 (b)에서는 *evaluation 3* 형태로 변경 되었는데 이는 SSA Form의 변환 과정 중에 적용 가능한 최적화인 상수 전파를 수행하였기 때문이다. 불필요한 표현식은 제거하고 단순히 상수 값으로 표현식을 나타낸 것이다.

<BL_15>의 문장 중에 *((Locali2_2 + Locali3_3) * Locali4_20)*은 <*BinExpr*> 표현식 (*Locali2_2 + Locali3_3*)을 포함한다. 이 경우도 위의 과정을 통해서 임시 변수인 *Locali6_34*로 대체된다. 모든 과정이 수정된 후에는 *evaluation (Locali5_9 := ((Locali2_2 + Locali3_3) * Locali4_20))*이 *evaluation (Locali5_9 := (Locali6_34 * Locali4_20))*로 대체된다. 즉, 표현식 *Locali2_2 + Locali3_3*이 임시 변수 *Locali6_34*로 대체된 것이다.

4. 실험 결과

실험은 펜티엄 4 2.4GHz, 메모리 512MB를 가진 PC에서 수행하였으며, 사용한 소프트웨어는 CTOC 작성과 테스트를 위해 eclipse 3.2을 사용하였고, 바이트코드 출력을 위해 editplus 2.11 버전을 사용하였다. 자바 컴파일러는 jdk1.5.0_09를 사용하였다.

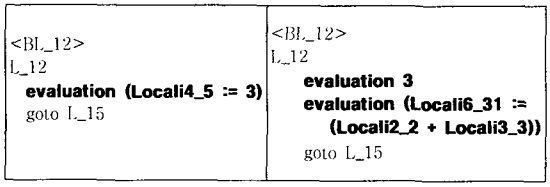


그림 7. (a)추가 전과 (b) 추가 후의 모습

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 6가지 경우를 **분석하였**다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문에서 사용한 예제를 이용하였다[16]. 표 1는 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 1의 예제를 이용해서 실험한 항목은 각 프로그램의 원시 소스의 라인 수, 바이트코드의 라인 수, 코드 변경 후 라인 수, 기본 블록 수, 간선의 수, 전체 노드의 수 등이다. 표 2는 eCFG로 변환한 후 생성된 정보에 대한 결과이다.

표 2에서 소스(no.)는 소스 코드의 라인 수를 의미하고, 바이트코드(no.)는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다. 변경 후(no.)는 CTOC를 통해 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 결과를 보면 바이트코드에 라벨 정보가 추가되기 때문에 기존의 소스보다 길이가 늘어나고 역 어셈블된 바이트코드보다는 길이가

표 1. 사용 예제와 간단한 설명

프로그램	설명
SquareRoot	숫자의 제곱근 찾기
SumOfSquare Roots	주어진 숫자 n에 대해 1부터 n까지 제곱근의 합 구하기
Fibonacci	주어진 숫자 n에 대해 피보나치 숫자인 Fn 찾기
BubbleSort	버블 정렬을 이용하여 정수 배열 정렬하기
LabelExample	라벨화된 break와 continue 프로그램
Exceptional	try-catch-finally 예외처리

표 2. eCFG 실험 결과

	소스 (no.)	바이트 코드 (no.)	변경 후 (no.)	기본 블록 (ea)	간선 (ea)	노드 (ea)
SquareRoot	37	94	60	15	18	99
SumOfSquare Root	38	103	63	18	19	108
Fibonacci	42	76	69	18	22	86
BubbleSort	30	79	68	16	21	101
LableExample	28	51	59	13	16	58
Exceptional	41	99	149	26	29	143

줄어드는 것을 확인할 수 있다. 기본 블록(ea)은 변경된 코드와 기본 블록을 위한 리더를 통해 생성된 기본 블록의 수를 의미한다. 간선(ea)은 기본 블록과 다른 기본 블록 사이의 관계를 표현하기 위해 사용된 간선의 수를 의미한다. 노드(ea)는 기본 블록 내에 명령어와 문장을 인식하기 위해 사용된 노드의 개수를 의미한다.

표 3은 정적 단일 배정 형태로 변환 후에 제어 흐름 그래프와 비교한 결과이다.

표 3의 경우 정적 단일 배정 형태로 변환된 이후 전반적인 라인수와 노드의 개수가 증가하는 것을 볼 수 있다. 이는 기존의 제어 흐름 그래프에 존재하지 않았던 ∅-함수의 추가에 의해서 발생하는 것이다.

표 4에서는 부분 중복 제거 적용 시 추가되는 P-문장의 수와 <*BinExpr*>의 개수를 확인하였다.

표 4의 P-문장 수는 기존의 SSA 과정에서 추가된 ∅-함수와 관련된 문장과 부분 중복 제거 과정에서 생성된 P-문장을 모두 합친 것이다. VN(Value Numbering)은 SSA 과정 후 타입 추론을 수행하는 과정에서 동일한 VN의 노드를 제거해서 표 3와 비교하면 노드의 개수가 줄어든 것을 확인할 수 있다.

표 3. 정적 단일 배정 형태 변환 후 결과

	CFG lines	SSA lines	%	CFG nodes	SSA nodes	%
SquareRoot	60	63	4.76	99	117	15.38
SumOfSquare Root	63	71	11.27	108	143	24.48
Fibonacci	69	77	10.39	86	126	31.75
BubbleSort	68	76	10.53	101	133	24.06
LableExample	59	63	6.35	58	74	21.62
Exceptional	149	177	15.82	143	304	52.96

표 4. 부분 중복 제거 테스트

	BinExpr	P-문장	VN	PRE
SquareRoot	7	3	117	115
SumOfSquare Root	9	8	129	125
Fibonacci	2	8	108	96
BubbleSort	5	8	117	147
LableExample	3	4	70	66
Exceptional	0	11	195	237

표 5는 기본적인 최적화인 복사 전파와 죽은 코드 제거를 수행한 결과와 부분 중복 제거 과정을 수행하여 표현식에 대한 최적화를 적용한 후 노드의 개수를 측정한 것이다. 추후 과정에서 생성된 노드에 복사 전파, 상수 전파 과정을 통해 최적화된 코드를 생성한다. 표 5는 최적화에 대한 테스트이다.

표 5를 살펴보면 첫 번째 SSA는 SSA Form으로 변환을 수행한 후 노드의 수를 나타내고 있다. 다음에 있는 COPY1과 DEAD1은 노드들에 대해 복사 전파와 죽은 코드 제거를 수행한 결과이다. 그 다음에 있는 PRE는 부분 중복 제거를 수행한 결과를 보이고 있다. PRE 결과 중에 BubbleSort와 Exceptional 예제의 경우에는 노드가 증가되는 것을 확인할 수 있는데 이 경우에는 부분 중복이 많이 발생되어 P-문장의 추가에 의해 노드가 증가되는 경우이기 때문이다. 하지만 이후 다시 복사 전파와 죽은 코드 제거를 수행하는 COPY2와 DEAD2과정을 통해 최적화된 코드를 얻을 수 있었다.

5. 결 론

요즘 많이 사용되고 있는 바이트코드는 스택 기반 코드이기 때문에 수행 속도가 느리고 프로그램 분석이나 최적화에 적절한 표현은 아니라는 단점이 존재한다. 따라서 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구되었다. 이를 해결하기 위해 최적화된 코드로 변환하는 CTOC를 구현하였다.

최적화 과정에서 CTOC는 정적으로 값과 타입을 결정하기 위해 변수를 배정에 따라 분리하는 SSA Form을 사용하였다. 하지만 기존의 eCFG에서 SSA

표 5. 최적화 테스트

	SSA	COPY 1	DEAD1	PRE	COPY 2	DEAD2
SquareRoot	117	117	117	115	113	109
SumOfSquare Root	143	143	129	125	123	117
Fibonacci	126	126	108	96	94	80
BubbleSort	133	133	117	145	145	103
LableExample	74	74	70	66	66	62
Exceptional	240	240	195	237	237	201

Form으로 변환하는 과정에서 ∅-함수의 삽입으로 인해 노드의 개수가 늘어나는 현상이 발생하였다. 이를 해결하기 위해 우선 SSA Form에서 복사 전파와 죽은 코드 제거 최적화를 적용하였다. 하지만 기존의 SSA Form은 표현식보다는 주로 변수에 관련된 것이었다. 따라서 좀 더 효율적인 최적화를 위해서는 SSA Form 형태의 표현식에 대해 중복된 표현식을 제거를 수행하였다. 일반적인 중복표현식 제거뿐만 아니라 부분 중복 표현식에 대한 제거를 수행하여 좀 더 효율적인 코드를 생성할 수 있었다. 또한 부분 중복 표현식 제거 후 다시 복사 전파와 죽은 코드 제거를 수행하여 더욱 효율적인 최적화가 수행되었다.

추후에는 CTOC에 좀 더 다양한 최적화를 적용하여 더욱 효율적인 최적화를 수행할 계획이다.

참 고 문 헌

[1] Tim Linholm and Frank Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison Wesley, Reading, MA, USA, Jan. 1997.

[2] James Gosling, Bill Joy, and Guy Steel, *The Java Language Specification*, The Java Series, Addison Wesely, 1997.

[3] Taiana Shpeismans and Mustafa Tikir, "Generating Efficient Stack Code for Java," *Technical report*, University of Maryland, 1999.

[4] 김기태, 유원희, "CTOC에서 자바 바이트코드를 이용한 제어 흐름 분석에 관한 연구," 한국콘텐츠학회 논문지, 제6권 제1호, pp. 160-169, 2006.

[5] 김기태, 유원희, "CTOC에서 자바 바이트코드를 위한 정적 단일 배정 형태," 정보처리학회논문지 D, 제 13-D권 제 7호, pp. 939-946, 2006.

[6] 김기태, 유원희, "정적 단일 배정 형태를 위한 정적 타입 배정에 관한 연구," 한국콘텐츠학회 논문지, 제6권 제2호, pp. 117-126, 2006.

[7] 김기태, 김지민, 김제민, 유원희, "CTOC에서 자바 바이트코드 최적화를 위한 Value Numbering," 한국컴퓨터정보학회논문지, 11권6호, pp. 19-26, 2006.

[8] SOOT project: http://www.sable.mcgill.ca/soot.

[9] Bloat project: http://www.cs.purdue.edu/s3/projects/bloat.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques and Tools*, Addison Wesley, 1986.

[11] Andrew W. appel, *Modern Compiler Implementation in Java*. CAMBRIDGE UNIVERSITY PRESS, pp. 437-477, 1998.

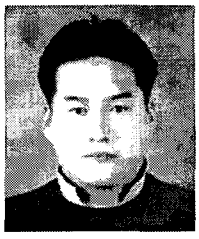
[12] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco. 1997.

[13] Morel, E and Renvoise, C. "Global optimization by suppression of partial redundancies," *Comm. ACM*, Vol. 22, No. 2, pp. 96-103. 1979.

[14] Knoop, J., Ruthing, O., and Steffen, B. "Optimal code motion: Theory and practice," *ACM trans. on Programming Languages and Systems*, Vol. 16, NO. 4, 1117-1155. 1994.

[15] Chow. F., Chan, S., Kennedy, R., Liu, S., Lo, R., and Tu, P. "A new algorithm for partial redundancy elimination based on SSA form," *In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. pp. 273-286. 1997.

[16] Don Lance, "Java Program Analysis: A New Approach Using Java Virtual Machine Bytecodes," http://www.mtsu.edu/~java.



김 기 태

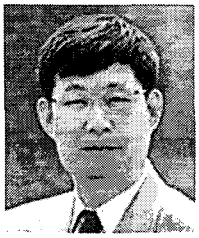
1999년 상지대학교 전자계산학과(학사)

2001년 인하대학교 전자계산공학과(공학석사)

2003년 인하대학교 전자계산공학과(공학박사수료)

2004년~2006년 인하대학교 컴퓨터공학부 강의전임강사

관심분야 : 컴파일러, 프로그래밍언어, 정보보안



유 원 희

1975년 서울대학교 응용수학과 (이학사)
1978년 서울대학교 대학원 계산학(이학석사)
1985년 서울대학교 대학원 계산학(이학박사)
1979년~현재 : 인하대학교 컴퓨터 공학부 교수
관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템