SSA Form에서 부분 중복 제거를 이용한 최적화

김 기 태[†] · 유 원 희[††]

요 약

CTOC에서는 정적으로 값과 타입을 결정하기 위해 변수를 배정에 따라 분리하는 SSA Form을 사용한다. SSA Form은 최근 데이터 흐름 분석과 코드 최적화를 위해 컴파일러의 중간 표현으로 많이 사용되고 있다. 하지만 기존의 SSA Form은 표현식보다는 주로 변수에 관련된 것이다. 따라서 SSA Form 형태의 표현식에 대해 최적화를 적용하기 위해 중복된 표현식을 제거한다. 본 논문에서는 좀더 최적화된 코드를 얻기 위해 부분 중복 표현식을 정의하고, 부분 중복 표현식을 제거하는 방법을 구현한다.

키워드 : CTOC, 자바 바이트코드, 제어 흐름 그래프, 정적 단일 배정 형태, 부분 중복 제거

Optimization Using Partial Redundancy Elimination in SSA Form

Kim Ki-Tae[†] · Yoo Weon-Hee[††]

ABSTRACT

In order to determine the value and type statically, CTOC uses the SSA Form which separates the variable according to assignment. The SSA Form is widely being used as the intermediate expression of the compiler for data flow analysis as well as code optimization. However, the conventional SSA Form is more associated with variables rather than expressions. Accordingly, the redundant expressions are eliminated to optimize expressions of the SSA Form. This paper defines the partial redundant expression to obtain a more optimized code and also implements the technique for eliminating such expressions.

Key Words : CTOC, Java Bytecodes, Control Flow Graph, Static Single Assignment Form, Partial Redundant Elimination

1. 서 론

자바 바이트코드에는 많은 장점이 존재하지만 스택 기반 코드이기 때문에 수행 속도가 느리고, 프로그램 분석과 최적화에 적절한 표현은 아니라는 단점이 존재한다[1, 2]. 따라서 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다[3]. 최적화된 코드로 변환하기 위해 CTOC(Classes To Optimized Classes) 프레임워크를 구현하였다[4, 5, 6, 7].

CTOC에서는 바이트코드의 분석과 최적화를 위해 가장 먼저 제어 흐름 분석을 수행하였다[4]. 또한 데이터 흐름 분석과 최적화를 위해서는 변수가 어디서 정의되었고, 어디서 사용되는지에 대한 정보를 알아야 한다. 이들 정보를 정의(def)와 사용(use)이라고 하는데 전통적인 컴파일러에서는 정의-사용 고리로 정의와 사용에 대한 정보를 유지한다[8, 9, 10]. 변수를 정적으로 다루기 위해서 변수는 정의와 사용에 따라 분리되어야 한다. 왜냐하면 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문이다. CTOC에서는 정의-사용 고리 대신 SSA Form(Static Single Assignment Form)을 사용하였다[5].

SSA Form에서 각 변수는 유일한 정의를 갖는다. 즉, SSA Form은 주로 변수에 관련된 것이기 때문이다. 따라서 SSA Form은 변수에 대해 상수 전파, 복사 전파 그리고 죽은 코드 제거와 같은 최적화를 수행하기에는 적절한 형태이지만 부분 중복 표현식 제거와 같은 표현식에 대한 최적화를 적용하기에는 적절하지 않다는 단점이 존재한다.

SSA Form으로 변경된 표현식에 대해 최적화를 적용하기 위해 본 논문에서는 부분 중복 표현식 제거 방법을 제안한다. 부분 중복 제거(partial redundancy elimination)는 Morel과 Renvoise에 의해 처음으로 제안된 최적화 기술이다[11]. 최근 부분 중복 제거 방법은 전역 최적화에서 가장 중요한 요소 중 하나가 되고 있다. 부분 중복 제거는 기존의 GCSE (global common subexpression elimination)과 LICM(loop-invariant code motion)을 결합한 기술이다. 또한 데이터 흐름 분석을 통해 프로그램에서 부분 중복을 제거하는 것이다.

최근까지 SSA Form은 기본적으로 변수에 관한 문제를 해결하는 것에 제한되었다. 왜냐하면 표현식에 대한 정의와 사용의 개념이 변수를 고려하는 경우보다 덜 명확하기 때문

※ 이 논문은 인하대학교의 지원에 의해서 연구되었습니다.
† 정 회 원 : 인하대학교 컴퓨터 공학부 박사과정
†† 종신회원 : 인하대학교 컴퓨터 공학부 교수
논문접수 : 2006년 9월 5일, 심사완료 : 2007년 2월 26일

이다. 따라서 SSA Form은 표현식에 대한 문제를 해결하기 위해 쉽게 적용될 수 없었다.

CTOC에서 SSA Form을 가진 표현식은 변수처럼 정의되어 있지 않기 때문에 부분 중복을 제거하는데 어려움이 발생하게 된다. 이를 해결하기위해 본 논문에서는 임시 변수인 t를 추가하였다. t_1는 $t_1 = a_1 + b_2$와 같이 표현식을 임시변수에 배정하기 위해 사용한다. 표현식을 임시 변수로 표현하였기 때문에 동일한 표현식이 존재하는 곳을 이 임시변수로 대체할 수 있고, 표현식이 제어 흐름의 병합지점에 도달할 때 기존에 SSA Form에서 새로운 이름을 가진 변수를 다루는 것과 마찬가지로 임시 변수에 대한 Ø-함수를 적용할 수 있게 된다. 즉 표현식에 대한 임시 변수 t_1을 위해 병합지점에 P-문장을 추가한 후 병합을 수행할 수 있게 되는 것이다.

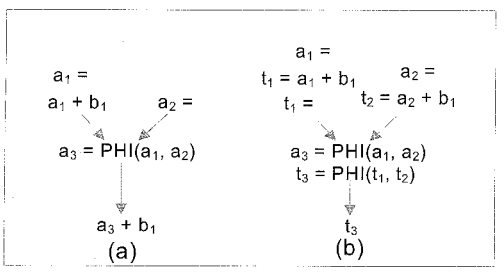
테스트 과정에서 병합지점에 표현식 선택을 위한 P-문장의 추가로 부분 중복 표현식 제거 과정에서 노드의 수가 늘어나는 경우도 존재하지만 이 후 상수 전파, 복사 전파 그리고 죽은 코드 제거 과정을 통해 더 효율적인 코드가 생성되는 것을 확인할 수 있다.

본 논문의 구성은 다음과 같다. 2장은 부분 중복 표현식의 의미와 관련 연구에 대해 살펴보고, 3장에서는 본 논문에서 사용할 예제와 SSA Form의 생성과정에 대해 설명한다. 4장에서는 부분 중복 제거 구현을 수행한다. 5장에서는 분석을 위해 필요한 그래프를 생성하고 결과 비료를 위한 실험을 수행한다. 마지막으로 6장에서는 결론과 향후 계획을 논한다.

2. 관련 연구

최적화 과정에서 동일한 표현식이 두 번 이상 평가되는 것은 바람직하지 않다. 만약 프로그램에서 a + b 표현식이 아무런 변화 없이 다시 사용된다면 뒤에 평가되는 식은 중복된 것이다.

(그림 1)의 (a)는 일반적인 SSA Form을 나타내는데 이 경우 Ø-함수를 표현하기 위해 P-문장인 *PHI(a_1, a_2)*를 사용하였다. 이때 P-문장은 a_1 또는 a_2와 같이 변수에 대한 정보만을 포함하였다. 하지만 본 논문에서 고려하는 부분 중복 표현식은 $a_1 + b_1$와 같은 표현식이 병합지점의 한쪽 선행자에서만 발생하는 경우이다. 이것에 대한 부분 중복 표현식을 제거하기 위해서는 처음 평가된 $a_1 + b_1$의 계산 결과를 임시변수 t_1을 사용하여 저장한 후, 다음에 나타나는 $a_1 + b_1$ 대신 이 임시 변수 t_1로 대체하면 중복된 계산을 피할 수 있게 된다. 즉, 부분 중복 제거는 제어 흐름 경로에서 불필요하게 발생하는 표현식의 중복된 계산을 제거하는 과정이라 할 수 있다. (그림 1)의 (b)와 같이 표현식에 대한 정의를 하나의 임시변수로 처리하고, 선행하는 다른 경로에 해당 표현식을 복사해주면 완전 중복 표현식 형태로 변형되기 때문에 기존의 SSA Form에서 사용하던 Ø-함수를 사용하는 방법을 적용할 수 있게 된다.



(그림 1) 부분 중복 표현식 제거

부분 중복 제거(partial redundancy elimination)는 Morel과 Renvoise에 의해 처음으로 제안된 최적화 기술이다[11]. 이후 Knoop et al.은 LCM(lazy code motion)이라고 불리는 배치 전략으로 불필요한 코드의 이동을 피하여 Morel과 Renvoise의 결과를 향상 시켰다[12]. 게다가 Knoop et al.은 술어 코드(predicated code)를 다룰 수 있도록 LCM 알고리즘을 확장하였다[13]. Hailperin은 LCM을 일반화 하였다[14]. 그의 일반화된 버전 역시 상수 전파와 연산 강도 경감을 수행하였다. Muchnick은 부분 중복 제거의 문제를 소개하고, 이 문제를 해결하는 고전적인 방법을 소개하였다[10]. 부분 중복 제거에 대한 대부분의 접근은 해결하고자 하는 문제를 비트 벡터와 데이터플로우 방정식 형태로 변형하고 반복적인 형태의 해결책을 제시하였다. Chow et al.는 SSAPRE라 불리는 새로운 접근법에 대해 소개하였다[15]. 그는 논문에서 SSA Form을 기반으로 하는 최적화 방법을 제안하였다. Kennedy et al.은 SSA-기반 프레임워크를 개발하였다[16]. 또한 Kennedy et al.는 연산강도 경감을 수행하기 위해 SSAPRE 알고리즘을 확장하였다[17]. 최근에는 speculative PRE 문제를 풀기 위한 최적화 알고리즘이 소개되고 있다[18, 19, 20].

3. SSA Form

본 논문에서는 CTOC의 부분 중복 제거 과정을 서술하기 위해 (그림 2)와 같은 부분 중복이 존재하는 예제 프로그램을 사용한다.

(그림 2)의 (a)와 (b)부분에 굵게 나타난 부분은 부분 중

```
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 2) (a) 소스 (b) 바이트코드

복된 표현식을 표시한 것이다. (그림 2)(a)의 소스를 SSA Form으로 변환하기 위해서는 우선 (그림 2)의 (b)와 같은 바이트코드를 CTOC의 입력으로 사용한다. 실제 CTOC의 입력은 (b)의 역 어셈블된 형태가 아닌 .class 파일이다. 하지만 여기서는 변경되는 부분을 보여주기 위해 역 어셈블된 코드로 표현하였다.

CTOC에서 가장 먼저 수행되는 과정은 바이트코드에 대한 제어 흐름 분석이다. 바이트코드의 특성 때문에 기존의 제어 흐름 분석 기술을 바이트코드에 적합하게 확장해야 한다. 우선 바이트코드에 라벨 정보를 추가하여 분석을 용이하게 하고 트리 형태의 중간 표현으로 각 문장을 표현하였다. 트리 형태의 변환을 효과적이고 체계적으로 하기 위해 (그림 3)과 같이 [4]의 BNF 코드를 확장하여 사용하였다.

(그림 3)에서 굵게 처리된 부분은 [4]에서 사용한 BNF에 부분 중복 표현식 제거를 수행하기 위해 새롭게 추가된 부분을 나타낸다.

제어 흐름 분석 후 데이터 흐름 분석과 최적화를 위해 변수가 어디서 정의되고, 어디서 사용되는지에 대한 정보가 요구된다. 따라서 정적으로 값과 타입을 결정하기 위해 변수를 배정에 따라 분리하였다. 이를 위해 CTOC에서는 SSA Form을 사용하였다[5]. (그림 4)는 SSA Form에 의해 ∅-함수가 추가된 CFG를 보여준다. 이 때 ∅-함수는 다양한 진입 경로를 가진 SSA 변수를 선택하기 위해 사용된다.

(그림 4)의 <BL_15>에서 *PHI(L_12 = Locali4_5, L_4 = Locali4_13)* 부분이 ∅-함수를 나타내는 것이다. 이 SSA Form은 부분 중복 제거 과정의 입력으로 사용한다. (그림 4)에서 *<BL_15 HD = L_27 NON>*은 기본 블록을 의미하고, BL_15는 기본 블록의 이름으로 블록 내에 처음 나타나는 레벨 정보로 블록 이름을 지정한다. 그 뒤에 나오는 HD는 루프가 있는 경우 헤더 정보를 기록하기 위해 사용되는 필드인데 특별히 루프 헤더가 존재하지 않는 경우라면 시작 블록인 처음 블록을 지정하도록 되어 있다. 그 뒤에 나오는 NON 역시 루프의 종류를 구별하기 위해 사용되는 필드이다. 기본 블록내의 문장에 대해서는 [4, 5]를 참조하면 된다.

```
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt | PHIStmt
LabelStmt → Label
InitStmt → INIT  LocalExpresssion[]
ExprStmt → eval  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 → Stmt := PHI (Expression, Expression )
Block → <block  Label>
Label → label_ Num
Expression → ConstantExpression| DefExpr | StoreExpression | BinExpr
BinExpr → ArithExpression
DefExpr → MemExp
StoreExpression → (  MemExpr  :=  Expression )
MemExpr → MemRefExpr | VarExpr
VarExpr → LocalExpr
ArithExpression → Expression Op Expression
LocalExpression → (Stack | Local ) Type  Num (_undef | _Num)
ConstantExpression → '  ID '  | Num
```

(그림 3) BNF의 일부

4. 부분 중복 제거 수행

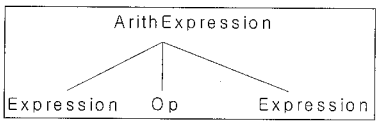
CTOC에 의해 생성된 SSA Form인 (그림 4)를 보면 <BL_4>와 <BL_15>에 표현식 *Locali2_2 + Locali3_3*이 중복된 것을 확인할 수 있다. 그러나 이 표현식은 <BL_4>와 <BL_15>에만 존재하고 선행자인 <BL_12>에는 존재하지 않기 때문에 전체 중복이 아닌 부분 중복 상태이다. 이것은 5장에 SSA Form 그래프인 (그림 12)를 통해 확인 가능하다. 기존의 경우에서는 위와 같은 경우 완전히 중복된 경우가 아니기 때문에 중복된 표현식에 대한 제거가 불가능 했지만, 본 논문에서는 분석을 통해 부분적으로 중복된 표현식을 중복이 존재하지 않는 선행 노드에 복사하는 과정을 통해 전체 중복으로 변환하여 부분 중복 표현식을 제거를 수행한다. 즉, 부분 중복된 표현식을 전체 중복으로 만들어 중복 표현식을 제거하는 것이다. 이를 위해 본 논문에서는 중복 가능한 표현식을 임시 변수로 대체하고 부분 중복된 표현식을 복사하여 병합지점의 선행자에 해당 표현식을 이동시켜 완전한 중복 형태로 변형한 후, SSA Form에서 사용된 것과 유사하게 임시 변수들을 병합 지점에서 P-문장에 의해 선택하도록 한다. 이런 과정을 수행하면 SSA Form으로 변환된 표현식에 대해서도 부분 중복을 처리할 수 있게 된다. 우선 SSA Form에서 중복된 표현식을 찾기 위해 가장 먼저 중복 가능한 표현식을 찾는 과정을 수행한다.

4.1 중복 가능한 표현식 찾기

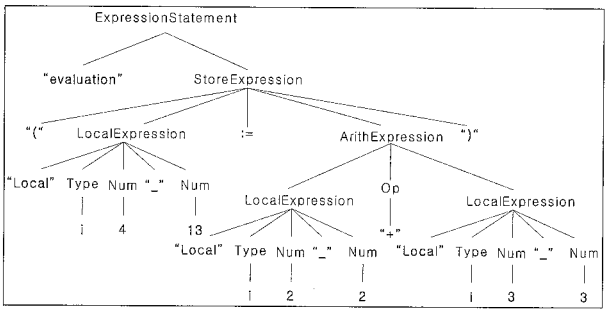
SSA Form에서 부분 중복 표현식 제거를 수행하기 위해 우선 문장 내에서 중복을 가질 수 있는 표현식을 먼저 찾아야 한다. 부분 중복을 가질 수 있는 a + b와 같은 표현식을 <*BinExpr*>라 정의한다. 예를 들면 <*BinExpr*>은 (그림 3)의 BNF에 있는 <*ArithExpression*> 형태를 의미한다. <*BinExpr*>이 될 수 있는 <*ArithExpression*>의 구

```
<BL_27 HD = null NON In>
    L_27

<BL_28 HD = L_27 NON INIT>
    L_28
        INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3
        goto L_0

<BL_0 HD = L_27 NON>
    L_0
        if0 (Locali1_1 ==0) then <BL_12 HD = L_27 NON> else <BL_4 HD = L_27 NON>

<BL_4 HD = L_27 NON>
    L_4
        evaluation (Locali4_13 := (Locali2_2 + Locali3_3))
        goto L_15

<BL_12 HD = L_27 NON>
    L_12
        evaluation (Locali4_5 := 3)
        goto L_15

<BL_15 HD = L_27 NON>
    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 HD = L_27 NON Out>
    L_29
```

(그림 4) 생성된 SSA Form



(그림 5) <ArithExpression> 구조



(그림 6) <ExpressionStatement> 구조

조는 (그림 5)와 같다.

(그림 5)의 <*ArithExpression*>은 하나의 <*Op*>와 두 개의 <*Expression*>을 갖는데 각 <*Expression*>은 트리에서 마지막 노드여야 한다. 마지막 노드가 되기 위한 조건은 각 <*Expression*>이 (그림 3)의 BNF에서 <*LocalExpression*>이거나 <*ConstantExpression*>이면 된다. <*LocalExpression*>은 바이트코드에서 사용되는 지역 변수를 의미하는 것이고 <*ConstantExpression*>은 상수를 의미한다.

제거 가능한 표현식을 찾기 위해 다음에 수행할 일은 동일한 표현식이 어느 곳에 나타나는가를 찾은 후 작업리스트에 추가하는 것이다. (그림 4)의 SSA Form 형태의 CFG를 통해 해당 블록과 각 문장을 방문하여 문장에 <*ArithExpression*>이 존재하는 가를 확인한다. 각 블록은 전위 순서로 방문한다. (그림 6)은 <BL*4*>의 <*ExpressionStatement*> 문장인 *evaluation (Locali4_13 := Locali2_2 + Locali3_3)*을 방문하는 경우를 나타낸다.

(그림 6)과 같은 트리 구조를 추적하는 중에 각 문장에서 <*Expression*>에서 상속된 표현식을 만나게 되면 해당 표현식이 <*BinExpr*>인가를 확인한다. 처음 방문한 문장인 <*ExpressionStatment*>는 <*Statement*>를 상속 받아 생성된 문장이기 때문에 <*BinExpr*>를 체크하지 않는다. 다시 <*ExpressionStatement*>의 자식인 <*StoreStatement*>를 방문하면, 이 경우에도 <*Statement*>클래스를 상속 받아 생성된 경우이기 때문에 역시 <BinExpr>를 고려하지 않는다. <*StoreStatement*>는 (그림 6)과 같이 :=의 좌측에 <*Local Expression*>과 우측에 <*ArithExpression*>으로 나누어진다. 좌측은 <*LocalExpression*> 형태만이 올 수 있는 위치이기 때문에 <*BinExpr*>를 고려하지 않아도 된다. 하지만 <*Expression*>의 경우에는 <*Expression*>으로부터 상속 받은 표현식들이 존재할 수 있는 위치이기 때문에 반드시 <*BinExpr*>에 대한 체크를 수행해야 한다. (그림 6)의 경우 <*Expression*>에 <*ArithExpression*>이 존재하는 경우이다. 하지만 모든 <*ArithExpression*>이 <*BinExpr*>인 것은 아니다. 예를 들면 $a+(b+c)$와 같이 중복된 경우도 존재하기 때문이다. 이러한 경우가 존재하는 가를 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 있게 된다. 하지만 왼쪽이나 오른쪽의 표현식이 또 다시 <*ArithExpression*>을 갖는 경우라면 현재의 표현식은 <*BinExpr*>가 아니다. 이렇게 생성된 <*BinExpr*>인 표현식은 다른 블록에서 중복된 표현식이 존재하는 가를 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. 모든 CFG를 방문한 후 생성된 작업리스트의 내용은 (그림 7)과 같다.

```
reals[0] = []
reals[1] = []
reals[2] = []
reals[3] = []
reals[4] = [(Locali2_2 + Locali3_3)]
reals[5] = [(Locali2_2 + Locali3_3)]
reals[6] = []
```

(그림 7) 작업리스트의 내용

(그림 7)의 reals[i]은 각 블록 별로 유지되는 <*BinExpr*> 정보를 유지하는 배열이다. i가 의미하는 것은 전위 순서로 SSA Form을 방문할 때 방문된 블록의 순서를 의미한다. 따라서 reals[4]와 reals[5]는 <BL_4>와 <BL_15>에 <*BinExpr*>이 존재한다는 것을 나타내며 내용도 함께 보여주고 있다.

4.2 부분 중복을 제거하기 위한 PHI-문장 위치 정하기

<*BinExpr*>가 존재하는 블록에 대한 정보를 작업리스트로부터 획득한 후 부분 중복 표현식 제거를 위해 P-문장을 SSA Form에 기반한 CFG에 삽입해야 한다. P-문장은 SSA Form의 ∅-함수와는 조금 다르다. SSA의 ∅-함수는 변수에 관한 것이지만 P-문장은 표현식과 관련된 것이다. 하지만 프로그램 내에서는 동일하게 <*PHIStmt*>로 처리한다. 왜냐하면 문장을 하나의 임시 변수로 변환한 후에는 SSA Form에서와 같이 변수에 관한 문제로 처리하면 되기 때문이다.

P-문장은 프로그램에서 표현식의 다른 값이 병합되는 지점에 도달할 때 요구된다. 다음 두 가지 경우엔 반드시 P-문장이 삽입해야 한다. 첫째, P-문장은 표현식이 존재하는 블록에 대한 반복적 지배자 경계가 되는 블록에 삽입된다. 둘째, P-문장은 반드시 표현식에 있는 변수들 중 하나를 위해 ∅-함수를 포함하는 블록에 삽입한다. 즉, 기존 CFG에서 ∅-함수가 존재하는 블록에 P-문장을 삽입하는 것이다. P-문장은 선행 노드에서 표현식에 대한 임시 변수가 정의 되고, 그 변수를 사용하는 병합노드에서 표현식에 대한 임시 변수를 정적으로 선택할 수 있다는 것을 나타낸다. (그림 4)에서 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 제거된다. 즉, <BL_15>이 P-문장이 삽입될 위치가 된다.

4.3 이름 바꾸기

표현식에 대한 재명명은 SSA Form에서 이름을 설정하기 위해 사용했던 재명명 방법과 유사하게 표현식에 대한 정보를 유지하기 위해 표현식 스택을 이용하여 CFG와 지배자 트리를 방문하면서 수행한다. 표현식 스택의 꼭대기는 표현식에 대한 정보를 유지한다.

재명명 과정은 우선 CFG의 기본 블록을 방문하여 해당 블록에 P-문장이 존재하는가를 확인한다. 또한 현재 블록에서 <*BinExpr*>형태의 표현식이 존재하는가를 확인한다. 만약 존재하는 경우라면 *Exist* 플래그를 *true*로 설정한다. 그리고 표현식 스택의 꼭대기에 이 *Exist*를 설정한다. *Exist*는 실제로 <*BinExpr*>이 사용된다는 것을 의미한다. 만약 표현식이 존재하지 않는다면 현재 노드의 후행자를 방문한다. 방문한 후행자에 P-문장이 존재하는 경우라면 P-문장의 피연산자 부분에 블록과 표현식 스택의 꼭대기에 설정한다. CFG의 후행자를 방문 한 후에는 지배자 트리에서 현재 블록의 자식을 방문한다. 현재 블록의 자식 블록에서 다시 앞에서 한 동작을 반복적으로 수행한다. 이 과정에서 재명명을 위한 또 다른 작업리스트를 사용한다. 재명명을 위한 작업리스트는 배열 리스트 형태로 생성한다. 예를 들면, <BL_15>에 초기에 추가되는 P-문장인 PHI(UD, UD)을 보면 <BL_4>를 방문한 경우 현재 블록에 *Locali2_2* + *Locali3_3*과 같은 표현식이 존재하는 경우이다. 이 표현식은 (그림 6)에서의 <*ArithExpression*> 노드이다. 해당 트리를 방문하고 표현식 스택의 꼭대기를 현재 표현식으로 설정한다. 표현식 스택의 꼭대기에 설정된 값은 표현식에 대한 정의를 나타낸다. 또한 이 값은 표현식의 배정을 유지하기 위해 사용된다. 만약 현재 블록의 후행 블록에 P-문장이 존재한다면 P-문장에서 해당 피연산자는 이 표현식 스택의 값으로 설정된다. 즉, P-문장의 해당 피연산자가 실제 변수로부터 배정 받는 다는 것을 의미한다. 만약 <BL_4>가 첫 번째 선행자로부터 배정되었다면 P-문장은 다음과 같이 PHI(1, UD) 형태로 표현된다. 즉, <BL_15>의 P-문장의 피연산자는 실제 표현식의 값을 사용한다는 의미이다. 반면 <BL_12>의 경우는 <BL_4>처럼 후행자에 P-문장을 갖지만 현재 블록에 *Exist*가 설정되지 않는 경우이다. 즉, 표현식 스택의 꼭대기에 아무런 표현식에 대한 정보를 지정할 수 없기 때문에 재명명 과정에서는 P-문장이 여전히 PHI(1, UD) 형태로 나타난다. 여기서 처음에 나오는 1은 첫 번째 선행자로부터 정보가 온다는 것을 표현하고, 이 값은 새로운 변수를 생성한 후에 Locali6_33과 같은 새로운 값으로 대체된다. 다음에 나오는 UD는 아직 정의되지 않았다는 의미이다. 즉, 완전히 중복되지 않고 여전히 부분 중복이 존재한다는 의미이다.

다음으로 지배자 트리에서 <BL_15>를 방문한 경우는, 현재 블록에 P-문장이 존재하는 경우이다. 이 경우는 표현식 스택의 꼭대기에 null 값을 설정한다. 하지만 블록의 P-문장은 정의 필드에 넣는다. 또한 <BL_15>는 현재 블록에 다시 *Exist*가 존재하는 경우이기 때문에 표현식을 방문하면서 정보를 얻는다. 만약 이때 정의 필드가 P-문장인 경우엔 현재 표현식을 재명명 작업리스트에 추가한다. 또한 해당 표현식을 스택의 꼭대기에 추가하여 중복된 표현식 값으로 설정하게 된다.

4.4 새로운 변수 생성

다음 과정은 부분 중복 제거 과정에서 사용할 새로운 변수를 생성하는 것이다. 현재까지 SSA Form 변환 과정에서 배정된 가장 큰 변수는 <BL_15>에서 사용되는 Locali5이기 때문에 새로운 변수는 Locali6_UD 형태로 생성 된다.

4.5 코드 이동하기

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(1, UD)인 상태이기 때문에 선행 블록 <BL_12>에 피연산자에 대한 정보가 존재하지 않은 상태이다. 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-문장의 피연산자를 생성하는 동작을 수행한다. 이 과정에서 임시로 생성된 표현식에 새로운 값 번호(VN : Value Number)를 설정한다[7]. 이 VN은 P-문장에서 사용될 임시변수 t와 같은 VN을 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] : 36은 추가된 문장에 설정된 새로운 값 번호를 함께 나타낸 것이다. 표현식을 임시변수에 배정하고 모든 노드에 같은 VN을 설정하게 된다. 생성된 문장은 표현식이 존재하지 않는 선행자의 마지막 <*JumpStatement*> 앞에 위치하게 된다. 즉 (그림 8) (b)와 같이 <BL_12>의 goto 문장 앞에 위치하게 된다.

(그림 8)의 (a)의 evaluation (Locali4_5 := 3) 문장이 (b)에서는 evaluation 3 형태로 변경 되었는데 이는 SSA 형태의 변환 과정 중에 적용 가능한 최적화인 상수 전파를 수행하였기 때문이다. 불필요한 표현식은 제거하고 단순히 상수 값으로 표현식을 나타내고 있다.

<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_34로 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_3이 임시 변수 Locali6_34로 대체된 것이다.

부분 중복 제거가 적용된 후 CFG는 (그림 9)와 같이 변경된다.

(그림 9)는 부분 중복 표현식 제거가 적용된 후 SSA Form 기반의 CFG를 나타낸다. (그림 9)에서 <BL_15>를 살펴보면, P-문장인 Locali6_34 := PHI(Locali6_33, Locali6_31)은 <BL_4>와 <BL_12>에 존재하는 표현식에 대해 임시 변수를 사용하고 <BL_15>에서는 각각의 임시 변수를 처리하고 SSA

<BL label_12>
label_12
evaluation (Locali4_5 := 3)
goto label_15
(a)

<BL label_12>
label_12
evaluation 3
evaluation (Locali6_31 := (Locali2_2 + Locali3_3))
goto label_15
(b)

(그림 8) 표현식 추가전과 후의 모습

```
<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 L_4>
   L_4
      evaluation (Locali4_13 := (Locali6_33 := (Locali2_2 + Locali3_3)))
      goto L_15

<BL_12>
   L_12
      evaluation 3
      evaluation (Locali6_31 := (Locali2_2 + Locali3_3))
      goto L_15

<BL L_15>
   L_15
      Locali6_34 := PHI(Locali6_33, Locali6_31)
      Locali4_20 := PHI(Locali4_13, 3)
      evaluation (Locali5_9 := (Locali6_34 * Locali4_20))
   L_23
      return Locali5_9

<BL L_29 EXIT>
   L_29
```

(그림 9) PRE 적용 후 SSA Form 형태의 CFG

Form으로 표현하기 위해 위와 같은 P-문장이 추가된 것이다.

이렇게 변경된 SSA Form을 이전에 SSA Form에서 수행했던 Value Numbering이나 복사 전파나 죽은 코드 제거를 수행하면 더욱 최적화된 코드를 얻을 수 있게 된다.

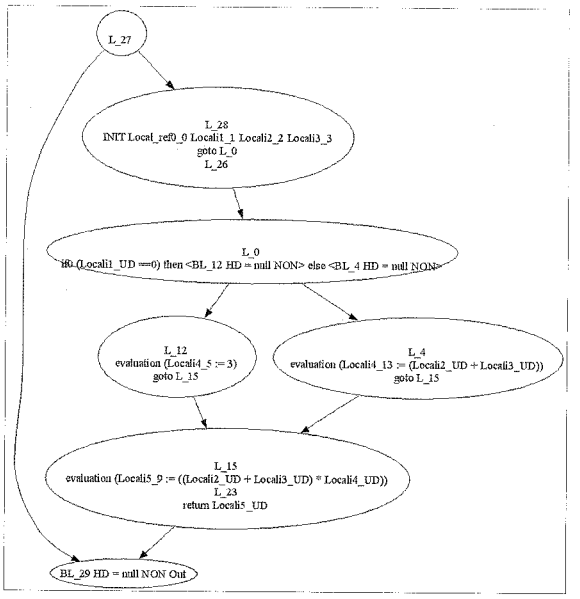
5. 실험

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. 또한 CFG와 SSA Form 변환 결과 확인을 위해 오픈 소스인 Graphviz를 사용하였다[21]. CTOC를 통해 필요한 정보를 .dot 파일로 생성한 후 결과를 jpg로 작성하였다.

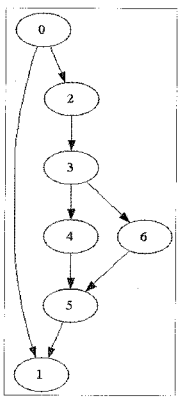
(그림 2)의 소스에 대해 작성된 CFG를 그림으로 작성한 것은 (그림 10)과 같다. 또한 (그림 11)은 깊이 우선 탐색에 대한 정보를 보여주는 그래프이다.

(그림 10)은 각 블록 별로 블록 내의 내용을 보여주며, 기본 블록간의 제어 흐름 관계를 보여준다. 반면 (그림 11)은 SSA Form으로 변환하는 과정에서 요구되는 깊이 우선 탐색 순서를 나타내는 그래프이다. 정보를 간단히 표현하기 위해 서로 다른 그래프를 작성하였다.

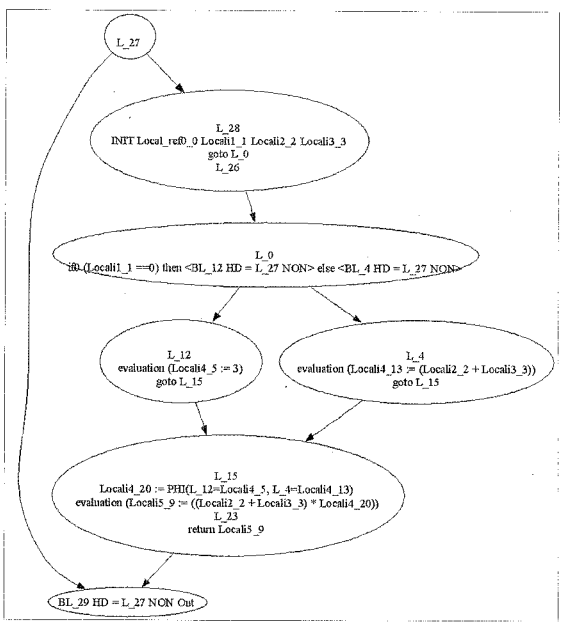
(그림 12)는 (그림 10)의 정보에 대한 SSA Form을 나타낸다. 역시 기존의 CFG에 SSA 관련 정보가 추가된 형태이다.



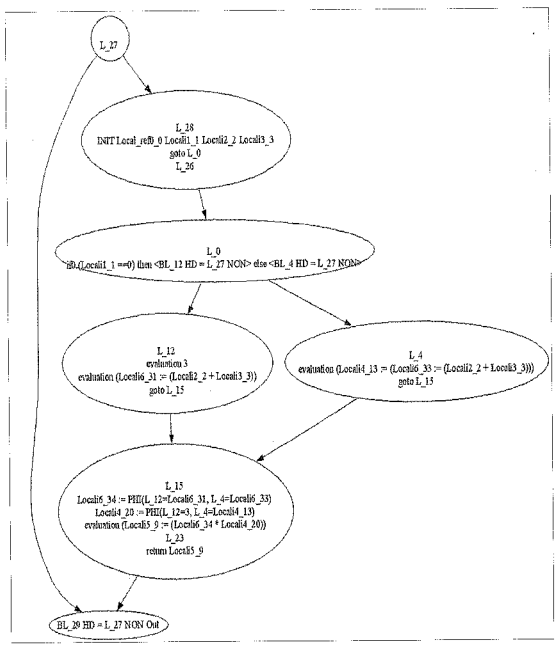
(그림 10) CFG



(그림 11) DFS



(그림 12) SSA Form



(그림 13) 부분 중복 제거 후

(그림 13)은 (그림 12)에 대해 부분 중복 제거를 한 후 나타나는 그래프를 보인다. (그림 12)와 (그림 13)을 비교해 보면 <BL_12>부분과 <BL_15> 부분에 변화가 생긴 것을 확인 할 수 있다. 이러한 이유에 대해서는 4.5절에서 설명하였다.

하지만 부분 중복 제거 후 다시 상수 전파, 복사 전파 그리고 죽은 코드 제거 과정을 다시 수행하여 좀 더 최적화된 코드를 얻게 된다.

<표 1>에서 TestPRE는 본 프로그램에서 사용된 예제이고 나머지 예제들은 Don Lance의 논문에 있는 예제 소스를 사용하였다[22].

<표 2>에서는 부분 중복 제거 적용 시 추가되는 P-문장의 수와 *<BinExpr>*의 개수를 확인하였다.

<표 2>의 결과에서 프로그램에서 사용되는 *<ArithExpression>*에 따라 *<BinExpr>*이 발생하는 것을 확인 할 수 있었으며, 또한 P-문장 수는 기존의 SSA Form 변환 과정에서 추가된 P-문장과 부분 중복 제거 과정에서 생성된 P-문장을 모두 합친 것이다. 부분 중복 제거 과정에서 이러한 P-문장의 증가에 의해 오히려 전체적인 노드의 개수가 늘어나는 경우도 나타났지만 이후 최적화 과정을 통해 노드의 개수를 줄일 수 있었다.

<표 3>은 SSA Form으로 변환한 후 적용 가능한 최적화인 복사 전파와 죽은 코드 제거를 수행한 결과와 부분 중복 표현식 제거 과정을 수행하여 표현식에 대한 최적화를 적용한 후 노드의 개수를 측정한 것이다.

<표 3>을 살펴보면 첫 번째 SSA는 SSA Form으로 변환을 수행한 후 노드의 수를 나타내고 있다. 다음에 있는 copy1와 dead2는 SSA Form 노드들에 대해 복사 전파와 죽은 코드 제거를 수행한 결과이다. 그 다음에 있는 PRE는 부분 중복 제거를 수행한 결과를 나타내고 있다. PRE 결과 중에 TestPRE, BubbleSort 그리고 Exceptional 예제의 경우에는 노드가 증가된 것을 확인할 수 있는데 이 경우는 앞에서 이야기한 것과 같이 부분 중복이 많이 발생되어 P-문장의 증가에 의해 노드가 증가되는 경우이다. 하지만 이후 다시 복사 전파 copy2수행 후 죽은 코드 제거 dead2를 수행하는 과정을 통해 최적화된 코드를 얻을 수 있었다.

〈표 1〉 사용 예제와 간단한 설명

프로그램	설명
TestPRE	논문에 사용된 예제
SquareRoot	숫자의 제곱근 찾기
SumOfSquareRoots	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 예외 처리

〈표 2〉 부분 중복 제거 테스트

	BinExpr 수	P-문장 수
TestPRE	2	2
SquareRoot	7	3
SumOfSquareRoot	9	8
Fibonacci	2	8
BubbleSort	5	8
LableExample	3	4
Exceptional	0	11

〈표 3〉 최적화 테스트

	SSA	copy1	dead1	PRE	copy2	dead2
TestPRE	51	51	43	51	51	49
SquareRoot	117	117	117	115	113	109
SumOfSquareRoot	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

6. 결 론

최근 여러 분야에서 자바 바이트코드가 중간 표현으로 사용되고 있다. 하지만 실행 속도가 느리고 분석하기 어렵다는 단점을 가진다. 따라서 빠른 실행 속도와 프로그램의 이해를 높이기 위해서는 최적화와 프로그램 분석이 요구되고 있다.

이를 위해 CTOC(Classes To Optimized Classes)를 구현하였다. CTOC에서는 바이트코드의 분석과 최적화를 위해 가장 먼저 제어 흐름 분석을 수행하였다. 정적으로 값과 타입을 결정하기 위해서는 변수를 배정되는 것에 따라 분리해야 하는데 이를 위해 CTOC에서는 정의-사용 체인 대신 SSA Form을 사용하였다. SSA Form은 최근 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 많이 사용되고 있다. 하지만 SSA Form은 주로 변수에 관련된 것이다. 따라서 표현식에 대한 최적화를 적용하기에는 기존의 형태는 조금 어색하다는 단점이 발생하였다. 이를 위해 부분 중복 제거 방법이 제안하고 구현하였다. 부분 중복 제거는 기존의 전역 부분 표현식 제거와 루프 불변 코드

이동 기술을 결합된 기술로 데이터플로우 분석을 통해 프로그램에의 부분 중복을 제거하는 것이다.

CTOC에서 표현식은 변수처럼 정의되지 않기 때문에 본 논문에서는 임시 변수를 추가하였다. 임시변수 t_1은 $t_1 \leftarrow a_1 + b_1$와 같이 표현식이 변수에 배정되는 것을 나타내도록 하였다. 그리고 표현식이 제어 흐름의 병합 지점에 도달할 때, 임시 변수를 위한 P-문장을 추가하여 병합하였다. 이는 부분 중복된 표현식을 완전한 중복으로 바꾸어 일반적인 부분 표현식 제거 문제로 변형하여 처리하도록 하였다. 완전한 중복으로 만들기 위해서 *<BinExpr>* 표현식을 발견한 후 이를 작업리스트에 유지하고, 표현식의 정보를 유지하기 위한 표현식 스택을 이용하여 각 표현식에 대한 정의를 지정하였다. 제어 흐름의 병합 지점에서 선행자 블록을 찾아 마지막 문장 바로 앞에 부분 중복된 표현식을 추가하여 완전한 중복 형태를 만들었다.

실험을 통해 현재 작성된 부분 중복 표현식 제거 수행 후 생성된 노드에 복사 전파, 상수 전파, VN, 그리고 죽은 코드 제거하기 과정을 통해 좀 더 최적화된 코드를 생성할 수 있었다. 또한 추후 연구에서는 이익과 비용의 문제를 적용하여 성능 평가를 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, 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 제13-D권 제7호, 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] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, 1986.

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

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

[11] Morel, E and Renvoise, C. "Global optimization by suppression of partial redundancies". Comm. ACM 22, 2(Feb.) pp. 96-103. 1979.

[12] Knoop, J., Ruthing, O., and Steffen, B. "Optimal code motion: Theory and practice". ACM trans. on Programming Languages and Systems 16, 4 (Oct.), pp.1117-1155. 1994.

[13] Knoop, J., Collard, J.F., and Ju, R. D. "Partial redundancy elimination on predicated code". In Proceedings of the 7th Static Analysis Symposium (SAS'00). pp.260-279. 2000.

[14] Hailperin, M. "Cost-optimal code motion". ACM Transactions on Programming Languages and Systems 20, 6, pp.1297-1322. 1998.

[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] Kennedy, R., Chow, F. C., Dahl, P., Liu, S. M., Lo, R., and Streich, M. "Strength reduction via SSAPRE". In Proceedings of the 7th International Conference on Compiler Construction. Springer-Verlag, New York. 144-158. 1998.

[17] Kennedy, R., Chan, S., Liu, S.M., Lo, R., and Tu, P. "Partial redundancy elimination in SSA form". ACM Transactions on Programming Languages and Systems 21, 3, 627-676. 1999.

[18] Cai, Q. and Xue, J. "Optimal and efficient speculation-based partial redundancy elimination". In 1st IEEE/ACM International Symposium on Code Generation and Optimization. pp91-104. 2003.

[19] Scholz, B., Horspool, R. N., and Knoop, J. "Optimizing for space and time usage with speculative partial redundancy elimination". In Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp.221-230. 2004.

[20] Xue, J. and Cai, Q. "Lifetime Optimal Algorithm for Speculative PRE", ACM Transaction on Architecture and Code Optimization, Vol.3, No.2, pp.115-155. 2006.

[21] http://www.graphviz.org/

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

김 기 태

e-mail : kkt@inha.ac.kr
1999년 상지대학교 전자계산학과(학사)
2001년 인하대학교 전자계산공학과 (공학석사)
2003년 인하대학교 전자계산공학과 (공학박사수료)
2004년 ~ 2006년 인하대학교 컴퓨터공학부 강의전임강사
관심분야 : 컴파일러, 프로그래밍언어, 정보보안

유 원 희

e-mail : whyoo@inha.ac.kr
1975년 서울대학교 응용수학과(이학사)
1978년 서울대학교 대학원 계산학 (이학석사)
1985년 서울대학교 대학원 계산학 (이학박사)
1979 ~ 현재 : 인하대학교 컴퓨터 공학부 교수
관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템