韓國 컴퓨터情報學會 論文誌
第11卷 第6號, 2006. 12

2006-11-6-1-3

CTOC에서 자바 바이트코드 최적화를 위한 Value Numbering

김 기 태*, 김 지 민**, 김 제 민**, 유 원 희***

Value Numbering for Java Bytecodes Optimization in CTOC

Ki-Tae Kim*, Ji-Min Kim**, Je-Min Kim**, Weon-Hee Yoo***

요 약

CTOC에서 SSA Form으로 변경된 표현식에 대해 최적화를 적용하기 위해선 중복된 표현식을 제거하여야 한다. 본 논문에서는 이를 위해 VN(Value Numbering)을 적용하였다. VN을 수행하기 위해서 우선 SSA Form 형태로 정보를 유지하는 SSAGraph를 생성하고, 그래프에서 동등한 노드를 찾은 후, SCC(Strongly Connected Component)를 생성한다. SCC를 통해 동등한 노드에 같은 valnum을 설정하게 된다. VN을 수행한 결과 SSA Form 수행 과정에서 추가되었던 많은 노드에 대한 제거를 확인할 수 있었다. 설정된 valnum은 최적화와 타입 추론에 적용된다.

Abstract

Redundant expressions must be eliminated in order to apply optimization for expressions in SSA Form from CTOC. This paper applied VN(Value Numbering) for this purpose. In order to carry out VN, SSAGraph must be first generated to maintain the information in the SSA Form, equivalent nodes must be found and SCC(Strongly Connected Component) generated. Equivalent nodes are assigned with an identical valnum through SCC. We could confirm elimiations for many nodes that added at SSA Form process after VN. The valnum can be applied in optimization and type inference.

▸ Keyword : CTOC, Value Numbering, 강 결합 연결요소(SCC), 최적화(Optimization)

• 제1저자 : 김기태
• 접수일 : 2006.10.24, 심사일 : 2006.12.05, 심사완료일 : 2006. 12.15
* 인하대학교 컴퓨터정보공학과 박사과정 ** 인하대학교 컴퓨터정보공학과 석사과정
*** 인하대 컴퓨터공학부 교수
※ 이 논문은 인하대학교의 지원에 의해서 연구되었습니다.

I. 서론

자바 바이트코드를 최적화된 코드로 변환하기 위해 CTOC가 제안되었다[7, 8, 9]. CTOC는 바이트코드의 분석과 최적화를 위한 프레임워크이다. CTOC에서는 바이트코드의 분석과 최적화를 위해 제어흐름분석을 수행하고, 정적으로 값과 타입을 결정하기 위해서 변수를 배정에 따라 분리하는 SSA Form을 사용한다. SSA Form은 최근 데이터흐름분석이나 코드 최적화를 위한 컴파일러의 중간표현으로 많이 사용되고 있다. 하지만 정보가 병합되는 지점에 ∅-함수가 추가되어 오히려 정보가 많아지는 단점이 발생하게 된다.

SSA Form으로 변경된 표현식에 대해 최적화를 적용하기 위해서는 중복된 표현식을 제거해야 한다. 본 논문에서는 이를 위해 VN(Value Numbering)을 적용하였다. VN은 두 가지 계산이 동일할 경우, 그들 중 하나를 제거하기 위해 적용되는 코드 최적화 기술 중 에 하나이다. VN에서 컴파일러는 표현식이 항상 같은 valnum 값을 생성하면 두 표현식에 같은 valnum을 배정하고, 중복이 발견되면 해당 표현식을 제거한다. VN의 계산은 수행되는 연산에 대해 관련된 기호 값으로 표현된다. 즉, 같은 기호 값을 가진 두 가지 계산은 동일한 표현식을 나타내는 것이다.

VN을 수행하기 위해서 우선 SSA Form으로 정보를 유지하는 SSAGraph를 생성하고, 그래프에서 동등한 노드를 찾은 후, 강 결합 연결 요소인 SCC (Strongly Connected Components)를 생성한다. SSAGraph의 SCC를 이용하기 때문에 기존의 해싱 방법이나 분할 방법보다 더 많은 동등한 노드를 찾을 수 있게 된다. SCC를 통해 찾은 동등한 노드에 같은 valnum을 설정하게 된다. 이렇게 설정된 valnum은 동일한 노드를 표현하기 때문에 최적화와 정적인 타입 추론 과정에서 유용하게 사용된다.

본 논문의 구성은 2장에서는 관련 연구에 대해 소개하고, 3장에서는 본 논문에서 사용할 예제와 VN 수행 과정 그리고 수행 과정 후 결과를 보인다. 4장에서는 간단한 실험을 수행하고, 5장에서는 결론 및 향후 계획을 논한다.

II. 관련 연구

VN이란 두 가지 계산이 동등할 경우, 그들 중 하나를 제거하기 위해 적용되는 방법 중에 하나이다. 동등함을 증명할 수 있는 일반적인 방법은 해싱과 분할이다[1]. Cocke와 Schwartz가 제안한 해싱 방법은 이해하고 구현하기 쉬운 반면 지역적이라는 단점이 존재하고, Hopcroft가 제안한 분할 방법은 전역적이지만 상수 전파와 같은 최적화 방법을 다루기에 쉽지 않다는 단점이 존재한다[2, 3].

초기 VN은 개별적인 기본 블록에서 지역적으로 수행되었다. 기본 블록 내에서 VN은 표현식을 분할하기 위해 해싱을 이용한다. 이 후 VN은 확장된 기본 블록에서 수행되기 위해 전체 프로시저에서 전역적인 형태로 수행된다. 전역적인 형태에서는 SSA Form의 프로시저를 요구한다. 전역 처리인 경우인 GVN(Global VN)은 Reif와 Lewis에 의해 시작되었다[4]. 이후 새롭고 덜 복잡하고 이해하기 쉬운 방법이 Alpern, Wegman, 그리고 Zadeck에 의해 제안되었다[5]. GVN은 변수의 일치에 대해 나타내는 표기법에 대한 논의이다. 기본 아이디어는 만약 연산자와 피연산 계산이 동일할 경우하면, 각각의 2개의 변수를 같은 것으로 만드는 것이다. 예를 들어 a와 b가 동일하다면, c = a + 1과 d = b + 1은 동일한 경우이다. SSA Form에서 GVN을 정확히 수행하기 위해서는 프로시저에 대한 수정이 요구되고, 그 후 결과 흐름 그래프의 값 그래프를 정의한다. 동일한 노드는 두 노드가 일치하는 값 그래프에서 최대 관계로 정의되어진다. 일반적으로 동일한 노드인 경우는 라벨이 상수이고 내용이 일치할 경우 그리고 같은 연산자를 가지고 피연산자가 일치할 경우이다.

III. Value Numbering

```
<Block label_27 ENTRY>
    label_27
<Block label_28 INIT>
    label_28
    INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3
    goto label_0
<Block label_0>
    label_0
    if0 (Locali1_1==0) then <Block label_12> else <Block label_4>
<Block label_4>
    label_4
    evaluation (Locali4_13 := (Locali2_2 + Locali3_3))
    goto label_15
<Block label_12>
    label_12
    evaluation (Locali4_5 := 3)
    goto label_15
```

```
<Block label_15>
    label_15
    Locali4_20 := PHI(label_12=Locali4_5, label_4=Locali4_13)
    evaluation (Locali5_9:= ((Locali2_2 + Locali3_3) * Locali4_20))
    label_23
    return Locali5_9
<Block label_29 EXIT>
    label_29
```

그림 1. SSA Form 예제
Fig 1. Example of SSA Form

본 논문에서는 CTOC에서 VN 과정을 서술하기 위해 <그림 1>과 같은 SSA Form을 사용한다. 각 기본 블록에 존재하는 문장들은 트리 형태로 존재한다[7, 8]. 사용된 트리를 표현 트리라 하는데 기본 블록 내부에서 각 명령어를 트리 형태의 문장으로 표현하기 위해 사용한다. 표현 트리는 커다랗게 추상 클래스인 Expr 클래스로부터 파생된 표현식과 역시 추상 클래스인 Stmt 클래스로부터 파생된 문장으로 나눠진다. <그림 2>는 표현 트리를 구성하는 BNF 일부를 나타낸다.

```
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt
LabelStmt → Label
InitStmt → INIT   LocalExpr[]
ExprStmt → eval   Expr
JumpStmt → GotoStmt | IfStmt | ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto   Block
IfZeroStmt → if0 (   Expr   (==   |!= |> |>= |< |<= )   (null|0 )   ) then Block
               else   Block
ReturnExprStmt → return    Exp
Block → <block   Label>
Label → label_ Num
Expr → ConstantExpr| DefExpr | StoreExpr
DefExpr → MemExp
StoreExpr → (   MemExpr   :=   Expr )
MemExpr → MemRefExpr | VarExpr
VarExpr → LocalExpr | StackExpr
LocalExpr → (Stack | Local ) Type   Num (_undef | _Num)
ConstantExpr → '   ID '   | Num F | Num L | Num
```

그림 2. BNF 코드 중 일부
Fig 2. Part of BNF

<그림 2>의 BNF를 통해 작성된 Expr과 Stmt의 파생 클래스를 생성한다. 이 두 종류의 클래스의 차이는 Expr로부터 파생된 클래스들은 값을 유지할 수 있는 값 필드를 가진다는 것이다. 또한 각 Expr은 타입을 표현하기 위해 타입 필드를 추가로 가진다.

3.1 SSA Graph 생성

SSAG(SSA Graph)는 CFG에서 표현식들이 서로 어떤 관계가 있는가를 나타내는 그래프이다. SSAG는 유향 그래프로 노드는 표현식을 나타내고 간선은 표현식의 피연산자를 나타낸다. 배정은 변수를 가진 노드의 라벨에 의해서 표현된다.

본 논문의 VN 구현은 SSAG의 SCC를 통해서 수행된다. 우선 <그림 1>에 대해 SSAG를 생성하는 과정에서 동등 관계에 있는 노드를 먼저 구한다. 동등한 노드를 구하기 위해서는 SSA Form을 가진 CFG의 기본 블록을 전위 순서로 방문하면서 동등한 노드를 찾는다. 기본 블록의 문장들은 트리 형태로 구성되어 있다. 따라서 기본 블록을 찾은 후 해당 트리를 방문하여 각각의 문장을 구성하는 노드를 살펴보게 된다. Visitor 클래스의 visitXXX() 메소드를 이용하여 트리에 해당하는 각 문장을 방문한다.

동등한 노드를 생성하기 위해서 고려해야 되는 경우는 visitPhiStmt(stmt), visitVarExpr (expr), visitStoreExpr(expr)과 같은 문장 또는 표현식을 방문하는 경우이다. <그림 3>은 visitVarExpr (expr)의 경우이다.

```
public void visitVarExpr(VarExpr expr) {
  if (! expr.isDef()) {
    VarExpr def = (VarExpr) expr.def();
    if (def != null) {
      mkEq(expr, def);
    }
  }
}
```

그림 3. visitVarExpr(expr) 의 경우
Fig 3. Case of visitVarExpr

예를 들어, <그림 1>의 <Block label_28 INIT>에 있는 InitStmt에 존재하는 변수들은 모두 선언된 변수이기 때문에 <그림 3>의 isDef()를 만족한다. isDef() 메소드는 표현식이 변수를 정의하는 문장인가를 확인하는 메소드이다. InitStmt의 경우에는 변수에 대한 정의가 존재하기 때문에 처음 수행에서는 동등한 노드를 생성하지 않는다. <Block label_0>의 IfZeroStmt에서 사용되는 Locali1_1 변수는 LocalExpr이고 이것은 VarExpr을 상속 받았기 때문에 SSAG에 존재하는 visitVarExpr에서 수행된다. 이때 변수는 현재 블록에 정의가 존재하지 않기 때문에 if (! expr.isDef())을 만족하여 다음 문장을 수행하게 된다. VarExpr def = (VarExpr) expr.def();은 이전에 정의된 것의 정의를 가져와서 def를 생성

하게 된다. def의 값이 현재 존재하지 않는다면 조상의 값을 가져와 정의하게 된다. 현재 def에는 Locali1_1을 갖게 된다. 결국 if (def != null)을 만족하기 때문에 mkEq(expr, def)를 수행한다. mkEq() 메소드는 현재 expr과 def를 동등한 것으로 만들어 준다. mkEq를 생성하는 알고리즘은 알고리즘 1과 같다. <알고리즘 1>은 equival() 메소드를 통해서 동등한 노드에 대한 정보를 얻는다.

알고리즘 1. 동등한 노드를 구하는알고리즘
Algorithm 1. Algorithm of find equivalance nodes

```
Input: node1, node 2 ∈ Node
Output: equiv ∈ HashMap
procedure mkEq(node1, node2)
begin
    Set s1 ← equival(node1);
    Set s2 ← equival(node2);
    if (s1 != s2)
      s1.addAll(s2);
      Iterator iter ← s2.iterator();
      while (iter.hasNext())
            Node n ← (Node) iter.next();
            equiv.put(n, s1);
      endwhile
    fi
end
procedure equival(Node node)
begin
    Set s ← (Set) equiv.get(node);
    if (s == null)
      s ← new HashSet(1);
      s.add(node);
      equiv.put(node, s);
    fi
    return s;

end
```

equival() 메소드는 노드를 입력으로 받아 동등한 내용이 존재한다면 해쉬 맵(HashMap)으로 된 equiv로부터 내용을 가져온다. 이때 node는 key이고 결과로 집합 s가 반환된다. 만약 s가 null 인 경우라면 새로운 해시 집합을 생성하고 그 집합에 현재 노드를 추가한다. 예를 들면 <Block label_28 INIT>에서 Locali1_1의 경우라면 {Locali1_1= [Locali1_1]} 와 같은 equiv를 생성하게 된다. 이때 Locali1_1은 key이고 [Locali1_1]은 value가 된다.

<알고리즘 1>에서 만약 s1과 s2가 다른 경우라면 s1과 s2를 s1으로 합친다. 예를 들면 <Block label_0>에서 Locali1_1의 경우라면 [Locali1_1, Locali1_1]과 같다. s2의 내용을 가져와 equiv.put(n, s1)에 의해 equiv에 추가한다. equiv의 key로 노드 n을 사용하고 value로 Set s1을 사용한다. equiv의 결과는 {Locali1_1= [Locali1_1, Locali1_1], Locali1_1 = [Locali1_1, Locali1_1]} 과 같아진다.

모든 것이 수행된 후 최종 equiv의 내용의 일부는 <그림 4>와 같다.

```
{
1.  (Locali4_13 := (Locali2_2 + Locali3_3))=
    [(Locali4_13:= (Locali2_2 + Locali3_3)), Locali4_13, (Locali2_2 +
   Locali3_3), Locali4_13],
    ...
9.  Locali4_13=
    [(Locali4_13:= (Locali2_2 + Locali3_3)), Locali4_13, (Locali2_2 +
   Locali3_3), Locali4_13],
10. Locali1_1=
    [Locali1_1, Locali1_1],
11. (Locali4_5 := 3)=
    [(Locali4_5 := 3), 3, Locali4_5, Locali4_5],
    ...
20. Locali1_1=
    [Locali1_1, Locali1_1],
21. Locali4_5=
    [(Locali4_5 := 3), 3, Locali4_5, Locali4_5],
22. Locali3_3=
    [Locali3_3, Locali3_3, Locali3_3]
}
```

그림 4. equiv 내용의 일부
Fig 4, Part of contents of equiv

<그림 4>에서 볼드체로 된 10번과 20번을 보면 Locali1_1=[Locali1_1, Locali1_1]으로 나타난다. 하지만 10번의 Locali1_1는 <그림 1>에서 <Block label_28 INIT>이고 20번의 Locali1_1은 <Block label_0>에 존재하는 노드를 나타내는 것이다. 이들은 서로 다른 위치에 존재하는 노드이다. 명확한 구분을 위해 idNum 필드를 이용해서 서로를 구별한다.

3.2 SCC 생성

CFG의 각 노드에 valnum을 설정하기 위해서는 CFG를 방문하면서 방문한 노드와 <그림 4>의 동등한 노드를 이용하여 SCC로 설정하고 SCC에 같은 valnum을 설정하는 과정을 반복적으로 수행한다.

SCC를 이용한 VN은 Tarjan의 SCC를 찾기 위한 깊이 우선 탐색 알고리즘과 결합하여 사용한다. 본 논문에서는 ArrayList 클래스를 이용하여 SCC를 생성한다. 트리를 방문하여 현재 노드와 동등한 노드를 SCC 리스트에 추가하여

해당 노드에 대한 SCC를 생성한다. 만약 <Block label_28 INIT>를 방문하여 Locali1_1을 방문한 경우라면 <그림 4>의 10번에 해당하는 경우이기 때문에 SCC는 [Locali1_1, Locali1_1]이 추가된 형태가 된다.

3.3 VN 설정하기

SCC가 생성된 후 다시 <Block label_28 INIT>를 방문하는 경우 각각의 노드에 대해 SCC를 찾아 같은 valnum을 설정해야 한다. <Block label_28 INIT>에 있는 2번째 지역변수인 Locali1_1의 경우는 앞의 과정에 의해 Locali1_1 = [Locali1_1, Locali1_1] 임을 알 수 있다. 하지만 [Locali1_1, Locali1_1]의 Locali1_1은 서로 다른 위치에 존재하는 것이다. 이것을 구별하기 위해서 각 노드를 구별할 수 있는 idNum 필드를 추가하여 각 노드에 서로를 식별할 수 있는 번호를 붙였다. 따라서 실제로 Locali1_1의 SCC로 내용을 살펴보면 [SCC] = Locali1_1 {3} Locali1_1 {10}로 나타낼 수 있다. 이때 {idNumi}는 각 노드를 식별하는 key 필드를 의미한다. 즉, 첫 번째 Locali1_1은 <Block label_28>의 INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3에 존재하는 노드를 의미하고 두 번째는 <Block label_0>에 존재하는 if0 (Locali1_1 == 0) then <Block label_12> else <Block label_4>의 Locali1_1을 의미한다.

초기의 각 노드의 valnum 필드에는 -1을 설정한다. SCC에 한 개의 요소가 존재할 경우에는 단순히 valnum 필드에 새로운 번호를 설정하고, SCC에 2개 이상의 노드가 존재하는 경우엔 각각의 노드에 같은 valnum을 설정해야 한다. 예를 들면, 첫 번째 SCC의 요소는 Locali1_1 {3} 으로 valnum이 -1로 설정된 상태이다. 이 노드에 valnum을 설정하기 위해 <알고리즘 2>를 적용한다.

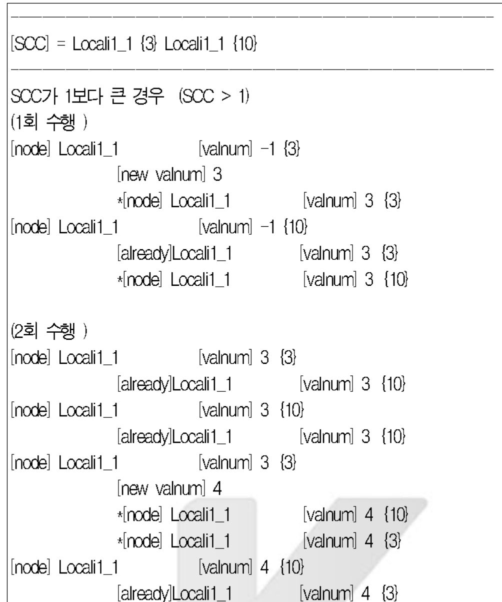
Locali1_1 {3}에 대해 알고리즘 2를 적용하면 Locali1_1 노드가 기존의 튜플에 존재하는가를 우선 확인한다. 알고리즘 2의 tuples는 HashMap으로 구성되는데 키 값으로 노드를 사용하고 내용으로 tuple인 <node, hash>를 갖는다. 예를 들어, <Block label_28 INIT>에 있는 Locali1_1은 기존의 tuples에는 존재하지 않았기 때문에 새롭게 tuple을 생성하여 tuples에 추가한다. 테이블로부터 tuple을 찾지만 현재 테이블에는 null인 상태이기 때문에 테이블로부터는 해당 노드를 찾을 수 없다. 따라서 현재 노드에는 새로운 valnum을 설정해야 된다. 기존의 value가 2까지 배정된 상태라 가정한다면 새로운 valnum을 설정하기 위해 value 값을 3으로 설정하게 된다.

알고리즘 2. VN을 설정하는 알고리즘
Algorithm 2. Set VN

```
Input: node ∈ Node, table ∈ HashMap
Output: changed ∈ Boolean
procedure VN(node, table)
begin
    boolean changed ← false;

    int temp ← -1;
    Tuple tuple ← (Tuple) tuples.get(node);
    if(tuple == null)
        Node s ← simplify(node);
        tuple ← new Tuple(s);
        tuples.put(node, tuple);        // {node = key, Tuple = value}
    fi
    Node w ← (Node) table.get(tuple);

    int value ← -1;
    if(w != null && w.getVN() != -1)
        value ← w.getVN() ;
    else
        value ← next++;
    fi
    Iterator iter ← ssaGraph.equivalent( node).iterator() ;
    while(iter.hasNext() )
        Node v ← (Node)iter.next() ;
        Tuple t ← (Tuple) tuples.get(v);

        if(t == null)
            continue;
        fi
        temp ← v.getVN() ;

        if(v.getVN() != value)
            v.setValueNumber( value);

            table.put( t, v);

            changed ← true;
        fi
    endwhile
    return changed;
end
```

다시 현재 노드와 동등한 노드가 있는가를 확인한다. 동등한 노드로 Locali1_1{10}와 자기 자신인 Locali1_1{3} 노드를 찾을 수 있다. 해당 노드를 기존의 tuples에서 찾는데 이 경우 이전 과정에서 Locali1_1{3} 노드를 tuples에 추가하였기 때문에 Locali1_1{3} 노드를 tuple로 구하게 된다. 또한 Locali1_1{3} 노드의 valnum을 -1로 설정했기 때문에 value 값과 다른 경우가 된다. 따라서 Lcoali1_1{3} 노드에 valnum을 현재 value 값인 3으로 설정하게 된다. 그리고 HashMap으로 구성된 table에 현재 구한 tu

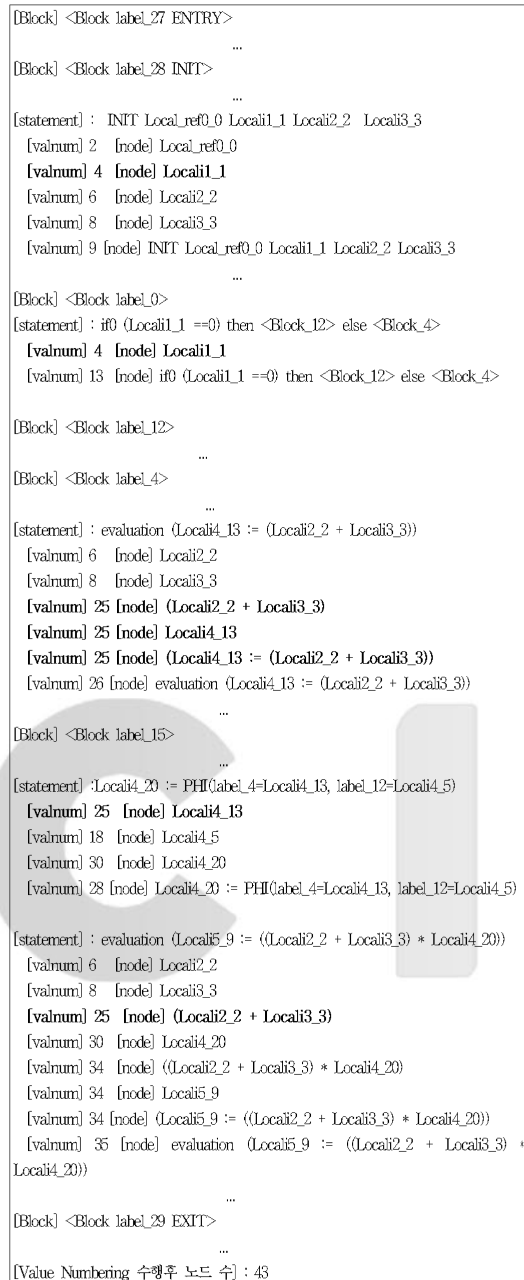
ple과 노드를 추가한다. 또한 현재 노드의 valnum에 변화가 생겼기 때문에 change 플래그를 true로 설정한다. 이 과정은 valnum필드에 더 이상 변화가 없을 때까지 수행된다. 다음 <그림 5>는 Locali1_1 {3}과 Locali1_1 {10}에 대해 VN이 수행되는 과정을 나타낸 것이다.



```
[SCC] = Locali1_1 {3} Locali1_1 {10}

SCC가 1보다 큰 경우 (SCC > 1)
(1회 수행 )
[node] Locali1_1          [valnum] -1 {3}
          [new valnum] 3
          *[node] Locali1_1          [valnum] 3 {3}
[node] Locali1_1          [valnum] -1 {10}
          [already]Locali1_1         [valnum] 3 {3}
          *[node] Locali1_1          [valnum] 3 {10}

(2회 수행 )
[node] Locali1_1          [valnum] 3 {3}
          [already]Locali1_1         [valnum] 3 {10}
[node] Locali1_1          [valnum] 3 {10}
          [already]Locali1_1         [valnum] 3 {10}
[node] Locali1_1          [valnum] 3 {3}
          [new valnum] 4
          *[node] Locali1_1          [valnum] 4 {10}
          *[node] Locali1_1          [valnum] 4 {3}
[node] Locali1_1          [valnum] 4 {10}
          [already]Locali1_1         [valnum] 4 {3}
```

그림 5. VN이 수행되는 과정
Fig 5. process of VN

<그림 5>는 VN 수행과정 중 일부를 출력물로 작성한 것이다. <그림 5>의 첫 부분은 Locali1_1과 관련된 SCC의 내용을 {idNumi}과 함께 출력한 것이다. valnum을 설정하는 과정에서 SCC가 2개 이상인 경우라면 valnum 설정에 변화가 없을 때까지 수행되어야 하는데 지금의 경우엔 2번 수행된 경우이다. 첫 번째 수행에서는 -1로 설정된 valnum을 해당 value 값으로 새롭게 설정하게 된다. 두 번째 과정은 기존에 각각의 노드에 대해서 valnum이 설정된 경우이기 때문에 [already] 필드가 설정되어 새로운 valnum을 배정하지 않는다. 두 노드 모두 [already]이기 때문에 더 이상 변화가 없어 valnum 설정이 끝난 상태이다. 하지만 다음 번호 배정을 위해 각각의 노드에 대해 3이 아닌 4값으로 변경하였다. 이는 다음에 수행되는 노드의 valnum 값을 4번이 아닌 5번으로 설정하기 위해서 수행한 것이다. 이 과정을 전체 프로그램에 적용한 결과는 <그림 6>과 같다.



```
[Block] <Block label_27 ENTRY>
                    ...
[Block] <Block label_28 INIT>
                    ...
[statement] : INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3
  [valnum] 2  [node] Local_ref0_0
  [valnum] 4  [node] Locali1_1
  [valnum] 6  [node] Locali2_2
  [valnum] 8  [node] Locali3_3
  [valnum] 9 [node] INIT Local_ref0_0 Locali1_1 Locali2_2 Locali3_3
                    ...
[Block] <Block label_0>
[statement] : if0 (Locali1_1 ==0) then <Block_12> else <Block_4>
  [valnum] 4  [node] Locali1_1
  [valnum] 13  [node] if0 (Locali1_1 ==0) then <Block_12> else <Block_4>

[Block] <Block label_12>
                    ...
[Block] <Block label_4>
                    ...
[statement] : evaluation (Locali4_13 := (Locali2_2 + Locali3_3))
  [valnum] 6  [node] Locali2_2
  [valnum] 8  [node] Locali3_3
  [valnum] 25 [node] (Locali2_2 + Locali3_3)
  [valnum] 25 [node] Locali4_13
  [valnum] 25 [node] (Locali4_13 := (Locali2_2 + Locali3_3))
  [valnum] 26 [node] evaluation (Locali4_13 := (Locali2_2 + Locali3_3))
                    ...
[Block] <Block label_15>
                    ...
[statement] :Locali4_20 := PHI(label_4=Locali4_13, label_12=Locali4_5)
  [valnum] 25  [node] Locali4_13
  [valnum] 18  [node] Locali4_5
  [valnum] 30  [node] Locali4_20
  [valnum] 28 [node] Locali4_20 := PHI(label_4=Locali4_13, label_12=Locali4_5)

[statement] : evaluation (Locali5_9 := ((Locali2_2 + Locali3_3) * Locali4_20))
  [valnum] 6  [node] Locali2_2
  [valnum] 8  [node] Locali3_3
  [valnum] 25  [node] (Locali2_2 + Locali3_3)
  [valnum] 30  [node] Locali4_20
  [valnum] 34  [node] ((Locali2_2 + Locali3_3) * Locali4_20)
  [valnum] 34  [node] Locali5_9
  [valnum] 34 [node] (Locali5_9 := ((Locali2_2 + Locali3_3) * Locali4_20))
  [valnum] 35 [node] evaluation (Locali5_9 := ((Locali2_2 + Locali3_3) *
Locali4_20))
                    ...
[Block] <Block label_29 EXIT>
                    ...
[Value Numbering 수행후 노드 수] : 43
```

그림 6. VN 수행 결과
Fig 6. Result of VN

<그림 6>을 보면 <Block local_28 INIT>의 Locali1_1과 <Block local_0>의 Locali1_1이 같은 valnum값 4를 갖는 것을 볼 수 있다. 또한 valnum값 25에 대해서 보면 <Block la

bel_4>와 <Block label_13>에 존재하는 것을 볼 수 있다. 특히 (Locali2_2 + Locali3_3)과 Locali4_13은 같은 valnum을 갖기 때문에 최적화 과정에서 대체하여 사용하게 된다.

Ⅳ. 실험

실험은 펜티엄 4 2.4GHz, 메모리 512MB를 가진 PC에서 수행하였으며, 사용한 소프트웨어는 CTOC 작성과 테스트를 위해 eclipse 3.1을 사용하였고, 바이트코드 출력을 위해 editplus 2.11 버전을 사용하였다. 자바 컴파일러는 j2sdk1.4.2_03을 사용하였다.

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 6가지 경우를 분석하였다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문에서 사용한 것을 이용하였다[6]. <표 1>는 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 1. 사용 예제와 간단한 설명
Table 1. Example and simple explanation

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

<표 1>의 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 2>는 실험 결과이다. <표 2>에서 소스(no.)는 소스 코드의 라인 수를 의미하고, 바이트코드(no.)는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다.

표 2. 변환 결과
Table 2. Conversion results

	소스	바이트코드	변경후	기본 블록	간선	노드
SquareRoot	37	94	60	15	18	99
SumOfSquareRoot	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	151

변경 후(no.)는 CTOC를 통해 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 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 3>는 VN에 대한 테스트이다.

표 3. VN 테스트
Table 3. VN Test

	CFG	VN	copy	dead	opt. VN
SquareRoot	99	117	117	117	115
SumOfSquareRoot	108	143	143	129	125
Fibonacci	86	126	126	108	96
BubbleSort	101	133	133	117	113
LableExample	58	74	74	70	66
Exceptional	151	240	240	195	194

<표 3>을 살펴보면 두 번째 VN은 SSA 수행 후 노드의 수를 나타내고 있다. 다음에 있는 copy와 dead는 노드들에 대해 복사 전파와 죽은 코드 제거를 수행한 결과 이다. 마지막에 있는 opt.VN은 다시 최적화된 VN을 수행한 결과를 보이고 있다. 첫 번째 CFG의 노드 수보다 두 번째 SSA 수행 후 VN의 숫자가 증가되는 것은 SSA Form으로 변환하는 과정에서 ∅-함수와 관련된 문장이 추가되었기 때문이다. 복사 전파와 죽은 코드 제거 후 다시 VN을 수행한 결과 SSA Form 수행 과정에서 추가되었던 많은 노드에 대한 제거를 확인하였다.

Ⅵ. 결론 및 향후 과제

자바 바이트코드를 최적화된 코드로 변환하기 위해 CTOC가 제안되었다. CTOC에서 SSA Form으로 변경된 표현식에 대해 최적화를 적용하기 위해 중복된 표현식을 제거하여야 한다. 본 논문에서는 이를 위해 VN을 적용하였다.

VN을 수행하기 위해서 우선 SSA Form 형태로 정보를 유지하는 SSAGraph를 생성하였고, 그래프에서 동등한 노드를 찾는 과정을 수행한 후, 동등한 노드의 정보를 유지하는 강 결합 컴포넌트인 SCC를 생성하였다. SCC를 통해 동등한 노드에 같은 valnum을 설정하는 과정을 수행하였다. 이렇게 설정된 VN은 추후 좀 더 많은 최적화 과정과 타입 추론 과정에 적용할 것이다.

참고문헌

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

[2] Cocke, John and Jacob T. Schwartz. Programming Languages and Their Compilers: Preliminary Notes, Courant Inst. of Math. Sci., New York Univ., New York, NY, 1969

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

[4] Reif, John R. and Harry R.Lewis. Symbolic Evaluation and the Global Value Graph, pp. 104-118, POPL77

[5] Alpern, Bowen, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs, pp. 1-11, POPL88

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

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

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

[9] 김지민, 김기태, 김제민, 유원희, "바이트코드를 위한 정적 단일 배정문 기반의 정적 타입 추론", 한국컴퓨터정보학회논문지, 제11권 제4호, pp. 87-96, 2006.

저 자 소 개

김 기태

2001년 2월 : 인하대학교 전자계산공학과 석사
2001년 3월 ~ 현재 : 인하대학교 컴퓨터정보공학과 박사과정
2004년 3월 ~ 2006년 2월 인하대학교 컴퓨터공학부 강의전임강사
<관심분야> 컴파일러, 프로그래밍 언어, 정보보안

김 지민

2004년 ~ 현재 : 인하대학교 컴퓨터정보공학과 석사과정
<관심분야> 컴파일러, 최적화

김 제민

2006년 2월~ 현재 : 인하대학교 컴퓨터정보공학과 석사과정
<관심분야> 컴파일러, 최적화

유 원희

1978년 2월 : 서울대학교 계산학과 이학석사
1985년 2월 : 서울대학교 계산학과 이학박사
1979년 ~ 현재 : 인하대학교 컴퓨터정보공학부 교수
<관심분야> 컴파일러, 프로그래밍 언어, 병렬시스템