韓國 컴퓨터情報學會 論文誌
第12卷 第2號, 2007. 5.

2007-12-2-1-1

CTOC에서 죽은 코드 제거 구현

김 기 태*, 김 제 민**, 유 원 희***

Implementation of Dead Code Elimination in CTOC

Kim Ki-Tae*, Kim Je-min**, Yoo Won-Hee***

요 약

자바 바이트코드가 많은 장점을 갖지만, 수행 속도가 느리고 분석하기 어렵다는 단점이 존재한다. 따라서 자바 클래스 파일이 네트워크와 같은 실행 환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다. 최적화된 코드로 변환하기 위해 CTOC를 구현하였다. 정적으로 값과 타입을 결정하기 위해 CTOC는 변수를 배정에 따라 분리하는 SSA Form을 사용하였다. 또한 문장의 표현을 위해 트리 구조를 사용하였다. 하지만 SSA Form 변환 과정에서 ∅-함수의 삽입에 의해 오히려 노드의 수가 증가하게 되었다. 본 논문은 SSA Form에서 더욱 최적화된 코드를 얻기 위해 죽은 코드를 제거하는 과정을 보인다. 각 노드에 대해 새로운 live 필드를 추가하고 트리 구조에서 죽은 코드 제거 과정을 수행한다. 실험 결과를 통해 죽은 코드 제거 후 상당한 노드의 수가 줄어든 것을 확인할 수 있다.

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 implements CTOC. In order to statically determine the value and type, CTOC uses the SSA Form which separates the variable according to assignment. Also, it uses a Tree Form for statements. But, due to insertion of the ∅-function in the process of conversion into the SSA Form, the number of nodes increased. This paper shows the dead code elimination to obtain a more optimized code in SSA Form. We add new live field in each node and achieve dead code elimination in tree structures. We can confirm after dead code elimination though test results that nodes decreases.

▸ Keyword : CTOC, 정적 단일 배정 형태(Static Single Assignment Form), 최적화(Optimization), 죽은 코드 제거(Dead Code Elimination)

• 제1저자 : 김기태
• 접수일 : 2007.2.23, 심사일 : 2007.4.11, 심사완료일 : 2007. 5.3.
* 인하대학교 컴퓨터정보공학과 박사과정, ** 인하대학교 컴퓨터정보공학과 석사과정
*** 인하대학교 컴퓨터 컴퓨터공학부 교수

I. 서 론

바이트코드는 스택 기반의 코드이고, 인터프리트 되기 때문에 실행 속도가 느리고 프로그램 분석과 최적화에 적절하지 못하다는 단점이 존재한다[1, 2]. 따라서 실행 환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다[3].

기존의 바이트코드를 최적화된 바이트코드로 변환하기 위해 CTOC(Classes To Optimized Classes) 프레임워크를 구현하였다[4, 5, 6, 7, 8]. 최적화 도구인 CTOC는 바이트코드의 분석과 최적화를 위해 제일 먼저 제어 흐름 분석을 수행하였다. 그 후 데이터 흐름 분석과 최적화를 위해 변수가 어디서 정의되고, 어디서 사용되는지에 대한 정보를 지정하였다. 이를 위해 전통적인 컴파일러에서는 정의-사용 고리로 정의(def)와 사용(use)에 대한 정보를 유지하였다[9, 10, 11]. 하지만 CTOC에서는 변수를 정적으로 다루기 위해 변수를 정의와 사용에 따라 분리하였다. 왜냐하면 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값과 다른 타입이 지정될 수 있기 때문이다. 따라서 정적으로 값과 타입을 결정하기 위해 변수는 배정에 따라 분리해야 한다. 이를 위해 CTOC에서는 정의-사용 고리 대신 SSA Form을 사용하였다[5]. SSA Form에서 변수는 유일한 정의를 갖는다는 특징을 가진다. 또한 CTOC에서는 SSA Form으로 문장을 표현하기 위해 트리 구조를 가진 중간 코드를 사용하였다.

바이트코드를 SSA Form으로 변환하는 과정 중 병합지점에 Ø-함수의 삽입에 의해 노드의 수가 증가되는 문제점이 발생하였다. 따라서 본 논문에서는 증가된 노드의 수를 줄이기 위해 SSA Form에서 적용 가능한 최적화 중 하나인 죽은 코드 제거를 수행한다.

전통적인 DCE(dead code elimination)는 데이터흐름 분석을 적용하기 위해 비트 벡터를 이용하거나 작업리스트와 정의-사용 고리를 이용하여 수행되었다[9]. 하지만 본 논문에서는 SSA Form으로 변환된 코드에 대해 DCE를 효율적으로 적용하기 위해 트리 구조를 이용하고, 살아있는 정보를 유지할 수 있는 *live* 필드를 각 노드에 새롭게 추가한다. 그리고 트리의 각 노드를 방문하면서 죽은 코드를 제거하는 과정을 수행하여 이전에 증가되었던 노드를 줄인다. 마지막으로 최적화된 결과를 실험을 통해 확인한다.

본 논문의 구성은 다음과 같다. 2장에서는 DCE의 의미와 관련 연구에 대해 살펴보고, DCE에서 사용할 예제와 기존 CTOC의 SSA Form 그리고 BNF에 대해서 설명한다. 3장에서는 DCE 과정을 통해 최적화를 수행하는 과정에 대해 기술한다. 4장에서는 DCE에 대한 실험을 수행하고 5장에서는 향후 계획과 함께 결론을 맺는다.

II. 관련연구

2.1 DCE(Dead Code Elimination)

어떤 변수가 프로그램의 특정 지점 이후에도 계속 사용될 수 있다면 그 변수는 살아있는 것이고 그렇지 않은 경우라면 변수는 죽은 것이다. 또한 결코 사용되지 않을 값을 계산한 문장도 죽은 코드에 해당한다. 전통적인 컴파일러에서 복사전파가 수행된 후 DCE가 적용되는 예는 그림 1과 같다[9].

x:= t_3	x:= t_3	
a[t_2] := t_5	a[t_2] := t_5	a[t_2] := t_5
a[t_4] := x	**a[t_4] := t_3**	a[t_4] := t_3
goto B_2	goto B_2	goto B_2
(a) 복사 전파 전	(b) 복사 전파 후	(c) 죽은 코드 제거 후

그림 1. 복사전파와 죽은 코드 제거의 예
Fig 1. Example of copy propagation and DCE

그림 1(a)는 블록 내에서 복사 전파 수행 전 상태를 나타낸다. (b)는 복사 전파가 수행된 결과를 보인다. x := t_3의 문장에서 x대신 임시 변수인 t_3이 아래쪽에 있는 a[t_4] := x 문장에 복사되어 a[t_4] := t_3으로 대체 되었다. 이렇게 수행된 후 변수 x는 아래쪽에서 더 이상 사용되지 않는 죽은 코드가 된다. 따라서 x:= t_3은 죽은 코드 제거에 의해 (c)와 같이 제거된 최적화된 코드를 얻게 된다.

죽은 코드는 다음과 같이 정의된다. 첫째, 만약 변수가 정의된 곳으로부터 끝나는 지점까지 어떤 경로에서도 사용되지 않는다면 변수는 죽었다고 한다. 둘째, 만약 명령어로부터 어떤 실행 가능한 경로에서도 사용되지 않는 값을 계산하는 경우라면 명령어는 죽었다고 한다. 셋째, 만약 죽은 변수의 값이 지역 변수에 배정된다면, 그 변수를 배정받은 변수와 명령어는 역시 죽은 것이다.

일반적으로 최적화를 수행하는 과정에서 의도적으로 죽은 코드를 생성하지는 않지만, 프로그램은 보통 최적화 수행 전에 죽은 코드를 포함하는 경우가 존재한다. 또한 출력문장에 영향을 주지 않거나, 연산 강도 경감과 복사 전파 등의 수행 과정을 통해 해당 변수가 다른 변수로 대체되는

경우에도 죽은 코드가 발생될 수 있다.

전통적인 DCE 기법은 계산하는 모든 명령어를 마크하면서 시작된다. 만약 함수에 의해 출력되거나, 명확하게 반환되는 경우, 그리고 함수의 외부로부터 접근될 수 있는 저장 위치에 영향을 주는 경우라면, 그 값은 살아있는 것이다. 따라서 DCE 알고리즘은 반복적으로 수행되면서 살아있는 값을 이용하여 명령어를 마크하게 된다. 알고리즘을 수행한 후 마크되지 않은 명령어는 죽은 것이기 때문에 제거되야 한다. 또한 죽은 코드의 확인은 데이터흐름 분석으로 공식화될 수 도 있다. 또한 작업리스트와 정의-사용 고리를 이용하여 수행할 수도 있다.

본 논문에서는 정의-사용 고리를 사용하는 대신 SSA Form을 사용하고, 데이터흐름 분석을 사용하는 대신 트리 구조를 사용하여 변수에 대한 정보를 획득 하였다. 또한 그림 1과 같이 복사 전파를 DCE 이전에 수행하여 복사 전파를 통해 복사 문장을 죽은 코드로 바꾸어줄 수 있기 때문에 더욱 효율적으로 수행할 수 있도록 하였다.

2.2 SSA Form

본 논문에는 CTOC에서 DCE 과정을 서술하기 위해 그림 2(a)와 같은 간단한 자바 소스를 사용한다.

```
int f(){
    int a=1, b=2;
    a = a + 3;
    b = 4;
    return (b);
}
```

```
int f();
  Code:
   0:   iconst_1
   1:   istore_1
   2:   iconst_2
   3:   istore_2
   4:   iinc        1, 3
   7:   iconst_4
   8:   istore_2
   9:   iload_2
  10:   ireturn
```

그림 2. (a) 소스 (b) 바이트코드
Fig 2. (a) source (b) bytecode

그림 2(b)는 (a)의 소스를 javap -c 옵션을 이용해서 역어셈블된 바이트코드이다. 그림 2(b)의 코드를 SSA Form으로 변환한 CFG는 그림 3과 같다. 일련의 변환 과정은 [5]에 자세히 기술되어 있다.

```
<BL_12 ENTRY>
    label_12

<BL_13 INIT>
    label_13
        INIT Local_ref0_0
        goto label_0

<BL_0>
    label_0
        eval (Locali1_1 := 1)
        eval (Locali2_2 := 2)
    label_4
        eval (Locali1_4 := (Locali1_1 + 3))
    label_7
        eval (Locali2_5 := 4)
    label_9
        return Locali2_5

<BL_14 EXIT>
    label_14
```

그림 3. SSA Form이 적용된 CFG
Fig 3. CFG which is applied SSA Form

그림 3의 각 기본 블록에 존재하는 문장들은 트리형태로 존재한다. 논문에서 사용된 트리를 표현 트리(*expression tree*)라 하는데 기본 블록 내부에서 각 명령어를 트리 형태의 문장으로 표현하기 위해 사용한다. 표현 트리는 커다랗게 추상 클래스인 〈*Expression*〉 클래스로부터 파생된 표현식과 역시 추상 클래스인 〈*Statement*〉 클래스로부터 파생된 문장으로 나눠진다. 그림 4는 표현 트리를 구성하는 *BNF*의 일부를 나타낸다.

```
Statement → ExprStmt | InitStmt | JumpStmt | LabelStmt | PHIStmt
LabelStmt → Label
InitStmt → INIT  LocalExpression[]
ExpressionStatement → 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(Label = Expression,Label = Expression )
Block → <BL_Label>
Label → label_ Num
Expression → ConstantExpr| DefExpr | StoreExpression | ArithExpr
DefExpr → MemExpression
StoreExpression → ( MemExpression := Expression )
MemExpression → VarExpr
VarExpr → LocalExpr | StackExpr
ArithExpression → Expr Op Expr
LocalExpression → (Stack | Local ) Type  Num (_undef | _Num)
ConstantExpression → '  ID '  | Num
```

그림 4. BNF의 일부
Fig 4. Part of BNF

그림 4의 *BNF*를 통해 작성된 〈*Expression*〉과 〈*Statement*〉의 파생 클래스를 생성한다. 이 두 종류의 클래스의 차이는 〈*Expression*〉로부터 파생된 클래스들은 값을 유지할 수 있는 *val* 필드를 가진다는 것이다. 또한 각 〈*Expression*〉은 타입을 표현하기 위해 *type* 필드도 추가로 가진다.

III. 죽은 코드 제거하기 수행

3.1 각 노드에 *DEAD*와 *LIVE* 설정하기

CTOC에서 *DCE* 과정을 수행하기 위해서는 *CFG*의 모든 노드를 전위순서(*preorder*)로 방문하면서 죽은 노드를 확인 할 수 있는 *live* 필드에 *DEAD*로 초기 값을 설정한다. *DEAD*는 불리언 값으로 *false*를 의미한다. 본 논문에서는 *live* 필드에 해당 노드가 죽었다는 의미로 *false* 대신 명확한 의미를 위해 *DEAD*를 사용하고 *true* 대신 *LIVE*를 사용한다.

우선 모든 노드를 *DEAD*로 설정한 후 *DCE* 과정을 수행하기 전에 특정 조건에 맞는 문장과 표현식을 미리 *LIVE*로 다시 설정한다. 왜냐하면 *LIVE*로 미리 설정되는 문장과 표현식은 프로그램에서 항상 살아있는 경우이기 때문이다. 미리 *LIVE*로 설정되어야하는 경우는 다음과 같다. 첫째, 문장이 프로그램의 출력에 영향을 주는 경우, 둘째, 문장이 배정문이고 배정문 중 *l-value*에 해당하는 *lhs*가 살아있는 문장에서 사용되는 경우, 셋째, 문장이 조건문이고 제어에 의해 하나 또는 그 이상의 문장이 실행되는 경우이다.

그림 4의 BNF에서 초기화 문장을 의미하는 〈*InitStmt*〉는 항상 *LIVE*하다고 가정한다. 그리고 배정문을 의미하는 〈*StoreExpression*〉은 *lhs*가 〈*Local Expression*〉인 경우 *LIVE*하다고 설정한다. 또한 〈*IfStmt*〉, 〈*GotoStmt*〉, 그리고 〈*ReturnExprStmt*〉와 같이 분기되는 문장 역시 *LIVE*하다고 가정한다. 이러한 과정을 기술하기 위해 알고리즘 1을 사용한다.

알고리즘 1. *LIVE* 설정 알고리즘
Algorithm 1. Algorithm of set *LIVE*

```
Input : node ∈ Node
Output: node ∈ Node
procedure setLive(node)
begin
    if (node is StoreExpression)
        expr ← node
        if (expr.live() == DEAD)
            expr.setlive(LIVE)
        fi
        if (expr.lhs.live() == DEAD)
            expr.lhs.setlive(LIVE)
            if (expr.lhs is VarExpr)
                worklist.add(expr.lhs)
            fi
        fi
        if (expr.rhs.live() == DEAD)
            expr.rhs.setlive(LIVE)
            if (expr.rhs is VarExpr)
                worklist.add(expr.rhs))
            fi
        fi
    fi
    if (node is Expression)
        parent ← node.parent()
        if (parent is ExpressionStatement)
            node ← parent
        fi
    fi
end
```

알고리즘 1은 노드가 *live* 필드의 초기 값으로 *DEAD*를 가진 경우 DCE를 수행하기 전에 문장과 표현식을 확인하여 미리 *LIVE*를 설정하는 알고리즘을 나타낸다. *LIVE*를 설정하기 전에 연결리스트로 된 *worklist*를 먼저 생성한다. 각각의 노드를 방문하면서 알고리즘 1을 수행하게 된다.

알고리즘 2는 노드를 방문하는 알고리즘이다.

알고리즘 2. 노드 방문 알고리즘
Algorithm 2. Algorithm of node visit

```
Input : cfg ∈ CFG
Output: cfg ∈ CFG
procedure visit(cfg)
begin
    case : visitStoreExpression(expr)
        if (expr.lhs is LocalExpression)
            expr.rhs.visit(this)
        else
            visitExpr(expr)
        fi
    esac
    case : visitVarExpr(expr)
        if (expr.live() == DEAD)
            expr.setlive(LIVE)
            worklist.add(expr)
        fi
    esac
    case : visitExpr(expr)
        if (expr.live() == DEAD)
            expr.setlive(LIVE)
        fi
        expr.visitChildren(this)
```

```
    esac
    case : visitStmt(stmt)
        if (stmt.live() == DEAD)
            stmt.setlive(LIVE)
        fi
        stmt.visitChildren(this)
    esac
end
```

노드 방문은 CFG의 각 블록과 블록 내의 노드를 전위 순서로 방문하면서 수행된다.

그림 3의 경우, 미리 *LIVE*를 설정해야하는 경우인 〈*InitStmt*〉, 〈*GotoStmt*〉 그리고 〈*ReturnExpr Stmt*〉가 존재한다. 따라서 이들 각 노드의 *live* 필드를 *LIVE*로 설정한다. 또한 현재 노드가 변수를 나타내는 〈*VarExpr*〉인 경우엔 *worklist*에 해당 표현식을 추가한다. 따라서 그림 3에 대해 알고리즘 1과 알고리즘 2가 적용된 후 *worklist*에는 [*Local_ref0_0, Locali2_5*]가 설정된다. *Local_ref0_0*는 소스의 클래스 이름인 *Dead*를 의미하는 변수이고 *Locali2_5*는 출력으로 사용되는 값이기 때문에 이 두 값은 *worklist*에 추가되고 이 변수들과 관련된 노드들은 모두 *LIVE*로 설정된다.

3.2 죽은 노드 제거하기

트리로부터 죽은 노드를 제거하기 위해서는 다시 *CFG*의 각 문장을 전위순서로 읽어 들인다. 그 후 읽어드린 노드의 *live* 필드의 값을 확인한다. *live* 필드의 값이 *DEAD*라면 해당 노드를 트리로부터 제거해야 하고, 그 외의 경우는 살아있기 때문에 *LIVE*인 상태가 된다. 하지만 *DEAD*인 노드 중에 〈*LabelStmt*〉와 〈*JumpStmt*〉는 프로그램 내에서 분기에 관련된 정보를 나타내기 때문에 제거하지 않는다.

죽은 노드를 제거하는 예로 그림 3의 블록 〈BL_0〉의 경우를 살펴보면, 첫 번째 문장은 〈*Label Stmt*〉에 해당하기 때문에 *live* 필드의 값이 *DEAD*이지만 위에서 설명한 이유 때문에 제거하지 않는다. 두 번째 문장인 *eval(Locali1_1 := 1)*의 경우 *live* 필드의 값이 역시 *DEAD*인 경우이다. 이 경우는 문장이 〈*ExpressionStatement*〉에 해당한다. 따라서 기존의 문장 리스트에서 제거가 수행 되어야 한다. 그림 5는 〈*ExpressionStatement*〉의 구조를 나타낸다.

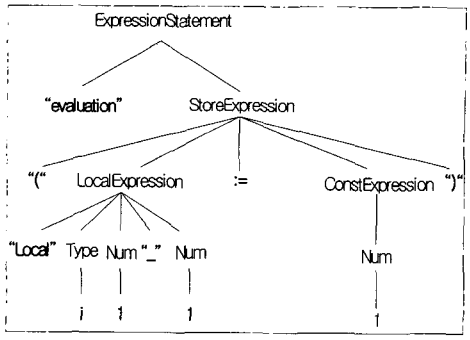

그림 5. 트리 구조
Fig 5. Tree structure

그림 5에서 만약 현재 노드가 루트인 〈*Expression Statement*〉인 경우라면 최상위 노드인 경우이다. 이 노드의 부모는 블록인 〈BL_0〉이다. 이 노드를 제거하기 위해서는 부모를 설정하는 setParent() 메소드에 *null*을 설정하여 현재 노드와 부모 노드의 연결을 제거한다. 또한 자식 노드인 〈*StoreExpression*〉을 다시 방문하여 이 노드의 부모인 〈*ExpressionStatement*〉와의 연결을 제거한다. 이것 역시 setParent(*null*)을 수행하면 된다. 그림 4에서 〈*StoreExpression*〉의 경우 구조를 보면 := 를 기준으로 왼쪽에 〈*Mem Expression*〉이 있고 오른쪽에 〈*Expression*〉이 올 수 있는 구조를 가진다. 〈*MemExpression*〉이란 메모리를 의미하는 것으로 배정문에서 좌측 값(*l-value*)을 의미하는 것이다. 본 논문에서는 좌측 값을 *lhs*로 나타내었다. 그림 4에서는 *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 〈*Expression*〉의 경우로는 〈*ConstantExpression*〉, 〈*DefExpr*〉, 〈*StoreExpression*〉, 그리고 〈*ArithExpression*〉 등이 올 수 있다.

다시 예제로 돌아가서 우선 *lhs*를 방문한다. 그림 5에서 현재 〈*StoreExpression*〉의 *lhs*인 *Locali1_1*은 〈*VarExpr*〉로부터 상속 받은 〈*LocalExpression*〉이다. 〈*LocalExpression*〉을 제거하는 경우에는 몇 가지를 고려해야 한다. 왜냐하면 〈*StoreExpression*〉의 *lhs*가 〈*LocalExpression*〉인 경우는 변수에 대한 정의가 발생하는 경우이기 때문에, 다른 문장에서 현재 정의된 변수를 사용하는 경우가 있을 수 있기 때문이다. 따라서 정의된 변수에 대한 사용 정보를 이용해서 현재 *lhs*를 제거할 때 다른 곳에서 사용되고 있는 노드를 함께 제거해야한다.

그림 3의 경우엔 〈BL_0〉의 *eval (Locali1_1 := 1)* 문장에서 *Locali1_1*의 정의 부분이 제거되면서 함께 아래 나오는 *eval (Locali1_4 := (Locali1_1 + 3))*의

Locali1_1 부분에 대해서도 제거가 수행되었다. 그림 6은 그림 3의 코드에 대해 *DCE*를 수행하고 난 후의 최종 결과이다.

```
<BL_12 ENTRY>
    label_12

<BL_13 INIT>
    label_13
        INIT Local_ref0_0
        goto label_0

<BL_0>
    label_0
    label_4
    label_7
        eval (Locali2_5 := 4)
    label_9
        return Locali2_5

<BL_14 EXIT>
    label_14
```

그림 6. 죽은 노드를 제거한 후 결과
Fig 6. Result of DCE

그림 6에서 보이는 것처럼 그림 3의 *eval (Locali1_1 := 1), eval (Locali2_2 := 2)* 그리고 *eval (Locali1_4 := (Locali1_1 + 3))* 문장도 역시 출력에 영향을 미치지 않기 때문에 제거되었다. 하지만 *eval(Locali2_5 := 4)*의 경우는 출력에 영향을 주는 경우이기 때문에 이미 앞의 수행과정에서 *LIVE*로 설정되어 제거되지 않는다. 또한 *label*과 *goto* 문장들은 분기와 관련된 정보 때문에 제거하지 않는다. 수행 결과 전체 노드는 복사 전파가 수행된 후 31개이었지만 DCE가 수행된 후에는 그림 6과 같이 많은 문장이 줄어들어 결국 노드가 17개로 줄어든 결과를 얻을 수 있었다. 좀 더 자세한 결과는 실험을 통해 확인한다.

IV. 실험

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

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 7가지 경우를 분석하였다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문의 예제를 이용하였다[12]. 표 1은 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 1. 사용 예제와 간단한 설명
Table 1. Examples and explanation

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

표 1에서 설명한 예제 프로그램에 대한 실험 항목은 각 프로그램의 원시 소스의 라인 수, 바이트코드의 라인 수, 코드 변경 후 라인 수, 기본 블록 수, 간선의 수, 전체 노드의 수 등이며, 표 2는 각 항목들과 실험 결과를 보여준다.

표 2. 실험 결과
Table 2. Result of experiment

	소스 (no.)	바이트 코드 (no.)	변경 후 (no.)	기본 블록 (ea)	간선 (ea)	노드 (ea)
Dead	8	14	12	4	4	28
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	143

표 2에서 소스(no.)는 자바 소스 코드의 라인 수를 의미하고, 바이트코드(no.)는 javap -c를 이용하여 생성된 바이트코드의 라인 수를 의미한다. 생성된 바이트코드는 자바 소스 코드를 스택구조에서 수행할 수 있도록 풀어논 형태이기 때문에 코드의 수가 증가하게 된다. 변경 후(no.)는 CTOC에서 기본 블록을 생성하기 위해 기존의 코드를 변경하는 과정에서 추가되거나 삭제된 후의 코드 라인 수를 의미한다. 일반적으로 바이트코드에 라벨 정보가 추가되기 때문에 기존의 소스보다 길이가 늘어나고 역어셈블된 바이트코드보다는 길이가 줄어드는 것을 확인할 수 있다. 기본 블록(ea)은 변경된 코드에서 리더를 통해 생성된 기본 블록의 수를 의미한다. 제어 흐름 그래프는 이 기본 블록을 통해 생성된다. 간선(ea)은 기본 블록과 다른 기본 블록 사이의 관계를 표현하기 위해 사용되는 간선의 수를 의미한다. 제어 흐름 그래프를 생성할 때 간선을 통해 선행관계와 후행

관계를 지정한다. 노드(ea)는 기본 블록 내에 명령어와 문장을 인식하기 위해 사용된 노드의 개수를 의미한다. 즉, 노드(ea)는 그림 5와 같이 문장을 구성하고 있는 노드의 개수를 의미한다. 이 정보는 각 노드를 방문하면서 카운트한 것이다.

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

표 3. 정적 단일 배정 형태 변환 후 실험 결과
Table 3. Result of SSA form

	CFG lines	SSA lines	%	CFG nodes	SSA nodes	%
Dead	15	15	0.0	31	31	0.0
SquareRoot	60	63	4.76	99	117	15.38
SumOfSquareRoot	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 3을 보면 정적 단일 배정 형태로 변환된 후 전반적으로 라인 수와 노드의 수가 증가된 것을 확인할 수 있다. 노드의 수가 증가된 이유는 기존의 제어 흐름 그래프에서는 존재하지 않았던 PHI-문장이 제어 흐름의 병합 지점에서 삽입되었기 때문이다. 즉, Ø-함수의 추가에 의해서 발생하는 것이다. 하지만 본 논문에서 사용한 예제인 Dead의 경우에는 프로그램 내에 병합되는 지점이 없었기 때문에 Ø-함수의 삽입이 발생하지 않아 표 3에서 보이는 것과 같이 노드의 추가가 발생하지 않았다.

표 4는 죽은 코드 제거를 수행한 후의 결과를 나타낸다.

표 4. 죽은 코드 제거 테스트
Table 4. Test of DCE

	CFG	SSA	copy	dead
Dead	31	31	31	17
SquareRoot	99	117	117	117
SumOfSquareRoot	108	143	143	129
Fibonacci	86	126	126	108
BubbleSort	101	133	133	117
LableExample	58	74	74	70
Exceptional	151	240	240	195

표 4를 살펴보면 본문의 예제인 Dead의 경우에는 기존에 SSA Form 수행 후 노드의 개수가 31인 것에 비해 죽은 코드 제거 후에는 17개로 변경된 것을 확인 할 수 있다. 왜냐하면 수행과정 중에 *live* 필드가 *DEAD*로 표현된 노드가 모두 제거되었기 때문이다. 표 4에서는 copy를 의미하는 복사 전파 수행 시 a = b = 1; 형태의 구문이 존재하지 않았기 때문에 복사 전파에 대해 커다란 효율을 발견할 수는 없었다. 하지만 a = b = c = d = 1; 과 같은 형태의 구문이 많이 존재한다면 복사 전파의 효율성은 극대화될 수 있다. 표 4의 dead 처럼 죽은 코드 제거를 통해 상당한 수의 노드가 줄어든 것을 확인할 수 있었다.

V. 결론 및 향후 계획

바이트코드는 스택 기반 코드이고 인터프리트 되기 때문에 수행 속도가 느리고 분석과 최적화에는 적절한 표현이 아니라는 단점이 존재한다. 따라서 본 논문에서는 네트워크와 같은 실행 환경에서 효과적으로 실행되기 위해 최적화된 코드로 변환을 수행하였다. 최적화 코드로 변환하기 위해 자바 바이트코드의 최적화 프레임워크인 CTOC를 구현하였다.

최적화 과정에서 CTOC는 변수를 배정에 따라 정적으로 분리하는 SSA Form을 사용하였다. 또한 문장의 표현을 위해 트리 구조를 사용하였다.

CTOC에서 죽은 코드 제거 과정을 수행하기 위해 가장 먼저 CFG의 모든 노드를 전위순서로 방문하면서 죽은 노드를 확인 할 수 있는 *live* 필드에 *DEAD*로 초기 값을 설정하였다. 다음으로 미리 *LIVE*를 설정할 수 있는 문장을 확인하여 *live* 필드를 *LIVE*로 설정하고 출력에 영향을 주는 변수를 *worklist*로 유지하며 해당 변수들과 관련된 문장들을 *LIVE*로 설정하였다. 이후 여전히 *DEAD*로 남아있는 노드들에 대해 죽은 코드 제거 과정을 수행하였다. 그 결과 기존의 노드에서 제거 가능한 노드를 발견할 수 있었고, 이 노드들을 제거하여 최적화된 트리를 생성할 수 있었다.

하지만 논 논문에서 사용한 SSA Form은 표현식보다는 주로 변수에 관련된 것이였다. 따라서 좀 더 효율적인 최적화를 위해서는 SSA Form의 표현식에 대해 중복된 표현식을 제거할 필요가 발생한다. 추후 중복 표현식 제거를 통해 좀 더 효율적인 코드를 생성할 수 있도록 연구를 진행할 것이다.

참고문헌

[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 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] 김지민, 김기태, 김제민, 유원희, "바이트코드를 위한 정적 단일 배정문 기반의 정적 타입 추론", 한국컴퓨터정보학회논문지, 제11권 제4호, pp. 87-96, 2006(6).
[8] 김기태, 김지민, 김제민, 유원희, "CTOC에서 자바 바이트코드 최적화를 위한 Value Numbering", 한국컴퓨터정보학회논문지, 11권6호, pp. 19-26, 2006
[9] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley, 1986
[10] Andrew W. appel, Modern Compiler Implementation in Java. CAMBRIDGE UNIVERSITY PRESS, pp. 437-477, 1998
[11] Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco. 1997.
[12] Don Lance, "Java Program Analysis: A New Approach Using Java Virtual Machine Bytecodes", http://www.mtsu.edu/~java

저 자 소 개

김 기태

1999년 2월 : 상지대학교 전산학과 이학사
2001년 2월 : 인하대학교 전자계산공학과 석사
2001년 3월 ~ 현재 : 인하대학교 컴퓨터정보공학과 박사
〈관심분야〉 컴파일러, 프로그래밍 언어, 정보보안

김 제민

2002년 2월 : 인하대학교 컴퓨터정보공학과 공학사
2006년 2월~ 현재 : 인하대학교 컴퓨터정보공학과 석사
〈관심분야〉 컴파일러, 최적화

유 원희

1975년 2월 : 서울대학교 응용수학과 이학사
1978년 2월 : 서울대학교 계산학과 이학석사
1985년 2월 : 서울대학교 계산학과 이학박사
1979년 ~ 현재 : 인하대학교 컴퓨터정보공학부 교수
〈관심분야〉 컴파일러, 프로그래밍 언어, 병렬시스템