韓國 컴퓨터情報學會 論文誌
第12卷 第1號, 2007. 3.

2007-12-1-1-1

CTOC에서 복사 전파

김 기 태*, 김 제 민**, 유 원 희***

Copy Propagation in CTOC

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

요 약

자바 바이트코드는 다양한 장점을 갖지만, 실행속도가 느리고 분석이 어렵다는 단점이 존재한다. 따라서 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다. 최적화된 코드로 변환하기 위해 CTOC가 구현되었다. CTOC는 기존의 바이트코드를 이용해서 CFG를 생성한 후 분석과 최적화를 위해 SSA Form을 생성하였다. 하지만 SSA Form으로 변환하는 과정에서 ∅-함수의 삽입으로 인해 노드의 개수가 늘어나는 현상이 발생하였다. 노드의 개수를 줄이기 위한 한 가지 방법으로 SSA Form에서 적용 가능한 최적화인 복사 전파를 수행하였다. 복사 전파란 하나의 변수 값이 다른 변수의 값으로 복사하는 과정이다. 복사 전파에 의한 변환은 변환 자체로는 큰 효과를 나타내지 못하는 경우가 존재하지만 이후 최적화 과정에서 변수가 사용되지 않는 경우 해당 변수에 대한 복사식을 제거할 수 있는 가능성을 제공하기 때문에 중요한 과정이다. 본 논문은 SSA Form에서 좀 더 최적화된 코드를 얻기 위한 복사 전파 수행을 보인다.

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. CTOC generated CFG using the existing bytecode then created the SSA Form for analysis and optimization. However, due to insertion of the ∅-function in the process of conversion into the SSA Form, the number of nodes increased. As a means of reducing the number of nodes, we performed copy propagation, which is an optimization method applicable to the SSA form. Copy propagation is the process of a value of a variable being copied to another variable. There are cases where conversion due to copy propagation alone does not yield significant effects. However, when variables are not used in the later optimization stages, copy propagation provides a means for eliminating the copy statement for the corresponding variable, making it an important step. This paper shows the copy propagation to obtain a more optimized code in SSA Form.

▸ Keyword : CTOC, 정적 단일 배정 형태(SSA Form), 최적화(Optimization), 복사 전파(Copy Propagation)

• 제1저자 : 김기태
• 접수일 : 2006.11.29, 심사일 : 2006.12.06, 심사완료일 : 2007. 3.15.
* 인하대학교 컴퓨터정보공학과 박사과정, ** 인하대학교 컴퓨터정보공학과 석사과정
*** 인하대학교 컴퓨터공학부 교수

Ⅰ. 서 론

바이트코드는 자바의 실행 코드로 많은 장점이 존재하지만 스택 기반 코드이기 때문에 수행 속도가 느리고 코드 분석과 최적화에 적절한 형태는 아니다[1, 2]. 네트워크와 같은 실행 환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다[3].

바이트코드를 최적화된 코드로 변환하기 위해 최적화 도구인 CTOC (Classes To Optimized Classes) 프레임워크를 구현하였다[4, 5, 6, 7, 8]. CTOC에서는 바이트코드의 분석과 최적화를 위해 가장 먼저 제어 흐름 분석을 수행하였다. 그 후 데이터 흐름 분석과 최적화를 위해 바이트코드를 SSA Form (Static Single Assignment Form)으로 변환하였다[11]. 왜냐하면 코드의 정적 분석을 위해서는 변수가 어디서 정의되고, 어디서 사용되는지에 대한 정보가 필요하기 때문이다. 동일한 변수라도 정의와 사용에 따라 다른 위치에서 다른 값과 다른 타입을 가질 수 있기 때문에 이러한 정보가 요구된다. 전통적인 컴파일러에서는 이를 위해 정의-사용 고리로 정의(def)와 사용(use)에 대한 정보를 유지한다[5].

정적으로 값과 타입을 결정하기 위해서는 변수는 배정에 따라 분리되어야 한다. 이를 위해 CTOC에서는 정의-사용 고리 대신 SSA Form을 사용하였다[5]. SSA Form은 최근 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 많이 사용되고 있다. 또한 많은 전역 최적화 알고리즘이 SSA를 기반으로 발전되고 있다. SSA Form은 이름처럼 모든 변수가 유일한 정의를 갖는 특징을 가진다.

기존의 바이트코드를 이용해서 CFG를 생성한 후 분석과 최적화를 위해 SSA Form으로 변환하는 과정에서 병합 지점에 ∅-함수의 삽입으로 인해 노드의 개수가 늘어나는 현상이 발생하였다. 노드의 개수를 줄이기 위한 한 가지 방법으로 SSA Form에서 적용 가능한 최적화인 복사 전파를 수행하였다.

일반적인 복사 전파는 데이터흐름 분석을 적용하기 위해 비트 벡터를 이용하거나 ACP(available copy instruction) 테이블을 이용해서 수행된다[12]. 하지만 본 논문은 SSA Form으로 변환된 코드에 대해 복사 전파를 효율적으로 적용하기 위해 CTOC의 트리 구조를 이용하고, 트리의 각 노드를 방문하면서 복사 전파를 수행하는 과정을 보인다.

복사전파에 의한 변환은 변환 자체로는 별다른 효과를 나타내지 못하는 경우가 존재하지만 이후 코드에서 변수가 사용되지 않는 경우 해당 변수에 대한 복사식을 삭제할 수 있는 가능성을 제공하게 된다[9].

본 논문의 구성은 다음과 같다. 2장은 복사 전파의 의미와 관련 연구에 대해 살펴본다. 또한 복사 전파에 사용할 예제와 기존 CTOC의 SSA Form과 BNF에 대해서도 설명한다. 3장에서는 복사 전파를 이용해서 최적화를 수행하는 과정에 대해 설명한다. 4장에서는 복사 전파에 대한 실험을 수행하고 5장에서는 결론과 향후 계획을 논한다.

Ⅱ. 관련연구

2.1 복사 전파

전통적인 컴파일러에서 x = y의 형태를 복사문 또는 복사라 한다. 복사 전파란 하나의 변수값이 다른 변수의 값으로 복사되는 문장 즉, 그림 1에서 x = y와 같은 배정문 S_1이 주어진 경우 이후 문장에서 다음 두가지 조건을 만족하는 경우 x가 사용되는 문장 S_2에서 y를 대신 사용해도 좋다는 개념이다[9, 10].

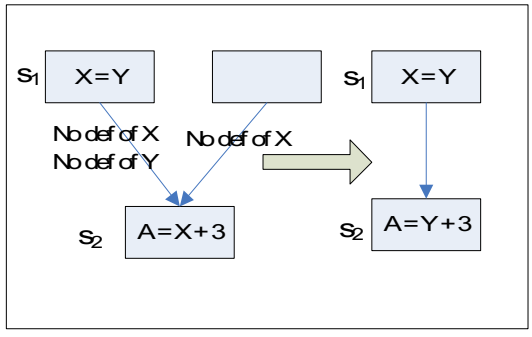

그림 1. 복사 전파 예제
Fig 1. Example of Copy Propagation

첫째, 문장 S_2에 이르는 경로 중 x의 값에 대해 정의되는 문장이 S_1 뿐이어야 한다. 즉, 다른 경로에서 혹은 S_1 문장과 S_2문장 사이에 x가 다른 값으로 정의되는 경우 x 대신 y를 사용한다면 원하지 않는 값이 사용될 가능성이 있기 때문이다. 두 번째, 문장 S_1에서 S_2로의 경로에 변수 y의 값에 대한 정의가 있어서는 안된다. 문장 S_1과 S_2사이의 경

로에서 y값이 변경된다면 x 대신 y를 사용하는 경우 잘못된 값이 사용되기 때문이다.

이러한 복사 전파에는 데이터흐름 분석을 이용하는 것이 일반적인 방법이다. Aho는 x = y에 대해서 사용-정의 고리 정보와 데이터 흐름 분석을 통해서 복사 전파를 수행하였다. Muchnick은 복사 전파를 지역과 전역으로 나누었다[12]. 지역적인 복사 전파는 분할된 기본 블록 내에서 수행되고, 전역 복사 전파는 전체 흐름 그래프를 통해 동작한다. 특히 ACP라 불리는 가능한 복사 명령어에 대한 테이블을 유지하면서 복사 전파를 수행한다. 복사 전파는 데이터흐름 분석을 적용하기 위해 일반적으로 비트 벡터를 이용해서 표현한다. 하지만 본 논문에서는 복사 전파를 위해 비트 벡터 대신 트리 구조와 사용(use)과 관련된 리스트를 이용한다. 왜냐하면 비트 벡터 정보나 ACP 테이블을 사용하게 된다면 불필요한 계산과 불필요한 메모리의 사용이 요구되기 때문이다. 하지만 CTOC에서 사용하는 트리 구조는 모든 정보를 노드 형태로 유지하기 때문에 더욱 효율적으로 처리할 수 있게 된다.

2.2 예제와 SSA Form

논문에서는 CTOC에서 복사 전파 과정을 서술하기 위해 그림 2와 같은 간단한 소스를 사용한다.

```
public class Copy {
        public void test(){
                int a, b , c;
                a = b = 1;
                c = a + b;
                System.out.println(c);
        }
}
```

그림 2. 복사 전파 소스
Fig 2. Source of Copy Propagation

SSA Form으로 변화는 과정 후에 CFG는 그림 3과 같다.

그림 3의 각 기본 블록에 존재하는 문장들은 3-주소 형태의 트리로 존재한다. 사용된 트리를 표현 트리라 하는데 기본 블록 내부에서 각 명령어를 3-주소 형태의 문장으로 표현하기 위해 사용한다. 표현 트리는 커다랗게 추상 클래스인 Expr 클래스로부터 파생된 표현식과 역시 추상 클래스인 Stmt 클래스로부터 파생된 문장으로 나눠진다. 그림 4는 표현 트리를 구성하는 BNF의 일부를 나타낸다[4].

```
<block label_17 ENTRY>
    label_17

<block label_18 INIT>
    label_18
    INIT Local_ref0_0
    goto label_0

<block label_0>
    label_0
    eval (Locali1_2 := (Locali2_1 := 1))
    label_4
    eval (Locali3_5 := (Locali1_2 + Locali2_1))
    label_8
    eval Ljava/lang/System;.out.println(Locali3_5)
    label_15

<block label_19 EXIT>
    label_19
```

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

그림 4의 BNF를 통해 Expr과 Stmt의 파생 클래스를 생성한다.

```
Stmt → ExprStmt | InitStmt | JumpStmt | LabelStmt | PHIStmt
LabelStmt → Label
InitStmt → INIT  LocalExpr[]
ExprStmt → eval  Expression
JumpStmt → GotoStmt | IfStmt | ReturnExprStmt
IfStmt → IfZeroStmt
GotoStmt → goto  Block
IfZeroStmt → if0 ( Expr (==  |!= |> |>= |< |<= ) (null |0 ))
              then Block  else  Block
ReturnExprStmt → return  Expression
PHIStmt → PHIJoinStmt
PHIJoinStmt → Stmt := PHI(Label= Expression, Label= Expresspression)
Block → <block  Label>
Label → label_ Num
Expression → ConstantExpression| DefExpr | StoreExpression |
ArithExpression
DefExpr → MemExp
StoreExpression → ( MemExpression := Expression )
MemExpression → VarExpr
VarExpr → LocalExpr | StackExpr
ArithExpression → Expression Op Expression
LocalExpression → ('Stack'|'Local') Type  Num (_undef | _Num)
ConstantExpression → '  ID '  | Num
```

그림 4. BNF의 일부
Fig 4. A part of BNF

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. 그림 4에서 굵게 표현된 〈StoreExpression〉부분은 복사 전파가 발생할 수 있는 BNF의 형태를 나타낸다.

III. 복사 전파 수행

복사 전파의 수행 단계의 첫 번째는 복사 전파가 발생할 수 있는 구조를 찾는 것이다. 이에 해당하는 구조는 그림 4의 BNF중 〈StoreExpression〉이다. 따라서 해당 구조를 찾은 후 본 논문에서 제시하는 복사 전파 알고리즘을 통해 복사 전파를 수행한다.

복사 전파 수행 과정은 다음과 같다. CTOC에 의해 생성된 SSA Form인 그림 3의 〈Block label_0〉를 보면 그림 2의 소스의 **int a = b = 1;** 부분이 **eval(Locali1_2 := (Locali2_1 := 1))**로 표현된다. 이 부분은 CTOC에서 〈Expression Statement〉로 표현된다. **eval(Locali1_2 := (Locali2_1 := 1))**을 트리 형태로 나타내면 그림 5와 같다.

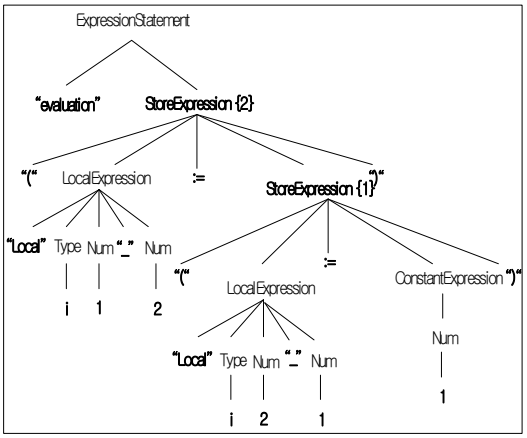

그림 5. eval(Locali1_2 := (Locali2_1 := 1))의 트리 표현
Fig 5. tree expression of eval(Locali1_2 := (Locali2_1 := 1))

복사 전파 과정은 그림 5의 트리의 각 노드를 전위 순서로 방문하면서 수행된다.

3.1 <StoreExpression>

복사 전파가 발생할 수 있는 구조를 살펴보면 그림 5의 트리 구조에서 〈StoreExpression〉가 이에 해당한다. 〈StoreExpression〉의 구조는 그림 6과 같다.

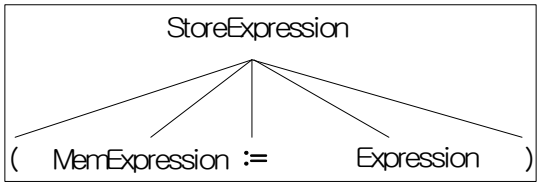

그림 6. 〈StoreExpression〉 구조
Fig 6. structure of 〈StoreExpression〉

그림 6의 〈StoreExpression〉의 구조를 보면 := 를 기준으로 왼쪽에 〈MemExpression〉이 있고 오른쪽에 〈Expression〉이 올 수 있는 구조를 가진다. 〈Mem Expression〉이란 메모리를 의미하는 것으로 배정문에서 좌측 값(l-value)을 의미하는 것이다. 본 논문에서는 좌측 값을 lhs로 나타내었다. 그림 4에서는 lhs로 〈VarExpr〉을 사용한다. 〈VarExpr〉은 일반적인 변수를 의미한다. 오른쪽은 배정문에서 우측 값(r-value)으로 rhs로 나타낸다. rhs로 〈Expression〉이 올 수 있는데 〈Expression〉의 경우로는 〈Constant Expression〉, 〈DefExpr〉, 〈StoreExpression〉, 그리고 〈ArithExpression〉 등이 올 수 있다. 특히 복사 전파에서는 〈StoreExpression〉의 우측 값에 다시 〈StoreExpression〉이 나타나는 경우를 고려한다. 이러한 경우가 복사 전파가 발생할 수 있는 경우이기 때문이다.

3.2 복사 전파 과정

그림 5에서 〈StoreExpression〉{1}과 〈Store Expression〉{2}는 복사 전파가 발생할 수 있는 경우이다. {1}과 {2}는 2개의 노드를 구별하기 위해 임의로 붙인 임시 번호이다. 우선 〈StoreExpression〉{1}의 경우는 (Locali2_1 := 1)을 나타낸다. 그림 6의 표현처럼 lhs로 Locali2_1이 오고 rhs로 1이 오는 경우이다. 이 경우는 단순히 변수에 대해 상수 값의 배정을 나타내는 것이다. 〈StoreExpression〉{2}의 경우는 (Locali1_2 := (Locali2_1 := 1))을 나타낸다. 이 경우는 복사 전파가 발생하는 경우이다. 복사 전파에 수행에 대해 알고리즘 1을 사용한다.

알고리즘 1. 복사 전파 알고리즘
Algorithm 1. Copy Propagation Algorithm

```
Input : expr ∈ StoreExpression
Output : changed ∈ Boolean
procedure copyProp(expr)

begin
  changed ← false
```

```
if (rhs instanceof StoreExpression)
    StoreExpression store ← (StoreExpression) rhs
    MemExpression rhsLHS ← store.lhs()
    Expr rhsRHS = store.rhs()

    if (rhsLHS instanceof LocalExpressjon)
      LocalExpression copy ← (LocalExpression) lhs.clone()
      copy.setDef(lhs)

      if (propExpr(expr.block(), (LocalExpr) rhsLHS, copy))
            changed ← true
            expr.visit(new ReplaceVisitor(rhs, rhsRHS))
            rhsLHS.cleanup()
            rhs.cleanupOnly()
      fi

      copy.cleanup()
    fi
  fi

  return change

end
```

알고리즘 1과 그림 5를 함께 살펴보면, (Locali1_2 := (Locali2_1 := 1))의 lhs는 Locali1_2 이고 rhs는 (Locali2_1 := 1)인 상태이다. 즉 현재의 rhs가 〈Store Expression〉인 상태이다. 따라서 알고리즘 1이 적용 가능하다. 현재 rhs를 store로 복사하고, store.lhs()로 rhs의 lhs를 새롭게 구해 rhsLHS에 저장한다. 그리고 store.rhs()로 rhs의 rhs를 rhsRHS에 저장한다. 이렇게 수행한 후 rhsLHS에는 Locali2_1이, rhsRHS에는 1이 저장된 상태이다. 다음은 rhsLHS가 〈Local Expression〉인가를 확인한다. 이때 이전에 생성한 lhs인 Locali1_2에 대한 복사본을 copy로 생성한다. 이 copy는 기존의 lhs와 동일한 성질을 갖는 새로운 노드이다.

propExpr(block, rhs, lhs)메소드를 통해서 복사 전파가 가능한가를 확인한다. propExpr (block, rhs, lhs)메소드는 해당 블록에서 rhs를 lhs로 전파하는 동작이 수행 가능한가를 확인하게 된다. 그림 5의 예제에서는 〈Block label_0〉에서 rhs에 해당하는 Locali2_1을 Locali1_2로 대체할 수 있는가를 확인한다. 실제 대체하는 동작은 ReplaceVisitor 클래스를 통해서 이루어진다. 이 클래스는 첫 번째 표현식을 두 번째 표현식으로 대체하는 동작을 수행한다. 즉, 이 클래스를 통해 실제로 Locali2_1을 Locali1_2로 대체하게 된다. 대체를 수행한 후 rhsLHS인 Locali2_1은 더 이상 필요 없는 노드이기 때문에 제거 가능하게 된다. rhs인 (Locali2_1 := 1)도 제거해야 한다. 왜냐하면 Locali2_1이 제거되었기 때문이다. 또한 그림 3에서 eval (Locali3_5 := (Locali1_2 + Locali2_1)) 문장도 Locali2_1이 Locali1_2로 대체되었기 때문에 eval (Locali3_5 := (Locali1_2 + Locali1_2)) 으로 변경되어야 한다. 이 동작 역시 ReplaceVisitor 클래스를 통해서 수행된다.

복사 전파와 관련된 모든 동작이 수행된 후 그림 3은 그림 7로 변경되어 진다.

```
<block label_17 ENTRY>
    label_17

<block label_18 INIT>
    label_18
    INIT Local_ref0_0
    goto label_0

<block label_0 header=null>
    label_0
    eval (Locali1_2 := 1)
    label_4
    eval (Locali3_5 := (Locali1_2 + Locali1_2))
    label_8
    eval Ljava/lang/System;.out.println(Locali3_5)
    label_15
    return

<block label_19 EXIT>
    label_19
```

그림 7. 복사 전파가 수행된 CFG
Fig 7. CFG at which copy propagation is performed

그림 7에서 〈Block label_0〉의 경우 그림 3에서 **(Locali1_2 := (Locali2_1 := 1))**부분이 복사 전파에 의해 **(Locali1_2 := 1)**으로 변경되었다. 이것은 Locali1_2와 Locali2_1이 같은 상수 값을 가지기 때문에 이 두 변수 중에 가장 나중에 값을 배정 받는 Locali1_2는 살리고 Locali2_1은 제거하였다. 이것은 일종의 상수 전파(constant propagation)를 수행한 것이다. **(Locali3_5 := (Locali1_2 + Locali2_1))** 문장도 Locali2_1 이 상수 전파에 의해 Locali1_2로 대체됨으로써 **(Locali3_5 := (Locali1_2 + Locali1_2))**로 복사 전파되는 것을 확인할 수 있다.

Ⅳ. 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 j2sdk1.5.0_07을 사용하였다.

예제 프로그램은 실험 결과의 비교를 위해 제어 흐름을 살펴볼 수 있는 7가지 경우를 분석하였다. 이 데이터들은 실험 결과의 비교를 위해 Don Lance의 논문에서 사용한 것을 이용하였다[13]. 표 1은 실험에 사용될 프로그램에 대한 간단한 설명이다.

표 1. 사용 예제와 간단한 설명
Table 1. test data and explanation

프로그램	설명
Copy	논문에 사용된 예제
SquareRoot	숫자의 제곱근 찾기
SumOfSquareRoot	주어진 숫자 n에 대해 1부터 n까지 제곱근의 합 구하기
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 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 실험 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. 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 2. 실험 결과
Table 2. Result of experiment

	소스 (no.)	바이트 코드 (no.)	변경 후 (no.)	기본 블록 (ea)	간선 (ea)	노드 (ea)
Copy	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
QuickSort	30	79	68	16	21	101
LableExample	28	51	59	13	16	58
Exceptional	41	99	149	26	29	143

표 2의 결과를 살펴보면, 바이트코드를 CTOC를 통해 변경한 후 라인 수가 줄어드는 것을 확인할 수 있다. 이것은 바이트코드에서는 여러 라인으로 표현되던 정보가 CTOC의 중간 표현에서는 간단히 표현되기 때문이다. 하지만 블록과 레벨에 대한 정보가 추가되기 때문에 고급 언어로 작성된 소스 보다는 더 많은 라인으로 표현되고 있다. 또한 실제 정보들은 트리 형태로 표현되기 때문에 많은 수의 노드가 생성되는 것을 확인할 수 있다. 이들 노드의 수가 줄어드는 것으로 최적화의 결과를 확인할 수 있다.

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

표 3. 정적 단일 배정 형태 변환 후 실험 결과
Table 3. Result after transformation

	CFG lines	SSA lines	%	CFG nodes	SSA nodes	%
Copy	14	14	0.0	28	28	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
QuickSort	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 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. 하지만 본 논문에서 사용한 Copy의 경우에는 프로그램 내에 병합되는 지점이 없기 때문에 ∅-함수의 추가가 발생하지 않아 코드가 증가하지 않는다.

표 4. Copy propagation 테스트
Table 4. Copy propagation Test

	CFG	SSA	COPY
Copy	28	28	26
SquareRoot	99	117	117
SumOfSquareRoot	108	143	143
Fibonacci	86	126	126
QuickSort	101	133	133
LableExample	58	74	74
Exceptional	151	240	240

표 4는 복사 전파를 수행한 후의 결과를 나타낸다. 표 4를 살펴보면 본문의 예제와 같은 경우에는 기존에 SSA Form 수행 후 노드의 개수가 28인 것에 비해 복사 전파 후에는 26개로 변경된 것을 확인할 수 있다. 왜냐하면 locali2_1과 (Locali2_1 := 1)을 나타내던 노드가 제거되었기 때문이다. 하지만 다른 예제의 경우에는 본문의 예제와 다르게 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 극대화 될 수 있다.

V. 결론

바이트코드를 네트워크와 같은 실행환경에서 효과적으로 실행되기 위해서는 최적화된 코드로 변환이 요구된다. 이를 위해 최적화 코드로 변환하는 CTOC 프레임워크가 구현되었다.

최적화 과정에서 CTOC는 정적으로 값과 타입을 결정하기 위해 변수를 배정에 따라 분리하는 SSA Form을 사용하였다. 하지만 기존의 CFG에서 SSA Form으로 변환하는 과정에서 ∅-함수의 삽입으로 인해 노드의 개수가 늘어나는 현상이 발생하였다. 이를 해결하기 위해 SSA Form에서 복사 전파를 수행하는 최적화를 적용하였다. 이를 위해 트리 구조에서 〈StoreExpression〉 노드를 찾고 복사 전파하는 알고리즘을 적용하여 최적화를 수행하였다. 그 결과 기존의 노드에서 제거 가능한 노드를 발견할 수 있었고, 이 노드들을 제거하여 최적화된 트리를 생성할 수 있었다.

복사전파에 의한 변환은 변환 자체로는 별다른 효과를 나타내지 못하는 경우가 존재하였지만 이후 코드에서 변수가 사용되지 않는 경우 해당 변수에 대한 복사식을 삭제할 수 있는 가능성을 제공하게 된다. 즉 복사전파는 기존의 SSA Form은 표현식보다는 주로 변수에 관련된 것이다. 따라서 좀 더 효율적인 최적화를 위해서는 SSA Form 형태의 죽은 코드 제거와 중복된 표현식에 대해 제거할 필요가 발생한다. 추후 죽은 코드 제거와 중복 표현식 제거를 통해 좀 더 효율적인 코드를 생성할 수 있도록 연구가 진행되어야 한다.

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 제13-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〕 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", Mar, pp 451-490, 1991
〔12〕 Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco. 1997.
〔13〕 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, 프로그램 분석

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