Dead Code Elimination in CTOC

Ki-Tae Kim, Je-Min Kim, and Weon-Hee Yoo
Department of Computer Science & Information Engineering, Inha University, Inchoen, KOREA
{kkt,jemin,whyoo}@inha.ac.kr

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 Φ-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.

1. Introduction

As stack-based code, byte code has such shortcomings as slow running speed and being inappropriate for program analysis and optimization[1, 2, 3]. This study constructed the **CTOC** (*Classes To Optimized Classes*) framework which is an optimization tool to convert byte code into optimized code. CTOC analyzed control flow first to analyze and optimize byte code. Then, to statically analyze variables, it was necessary to divide the variables according to definition and use because different values and types can be assigned to the same variable at different positions. For this purpose, CTOC used the SSA Form[6] instead of *du-chain*[4, 5]. Every variable of the SSA Form have only one definition. Moreover, CTOC used the *tree structure* as intermediate code to express statements in SSA Form.

However, one problem occurred in the process of converting byte code into SSA Form: the increase of nodes by the addition of the Φ-function at the merging point. To reduce the number of the increased nodes, this paper carried out **DCE**(*Dead Code Elimination*) which is one optimization method that can be applied in the SSA Form.

In CTOC, DCE is carried out by visiting each node of the tree. Traditionally, DCE used *bit vector*, or *worklist* and *du-chain* to apply data flow analysis. However, this paper used a *tree structure* to efficiently apply DCE to the code converted to SSA Form, and added a *live* field for storing liveness information to each node. Then, it visits each node of the tree again to remove dead codes so as to decrease the number of nodes that had been increased. Finally, the optimized result is checked through experiments.

The composition of this paper is as follows: Chapter 2 describes the examples to be used in DCE, the SSA Form of CTOC, and BNF. Chapter 3 describes the process of optimization through DCE. Chapter 4 describes the experiment on DCE, and Chapter 5 summarizes the conclusions.

2. SSA Form of CTOC

This paper uses the eCFG of the intermediate expression as shown in figure 1 to describe the DCE process in CTOC.

```
<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))
```

0-7695-2867-8/07 $25.00 © 2007 IEEE
DOI 10.1109/SERA.2007.102

```
    label_7
      eval (Locali2_5 := 4)
    label_9
      return Locali2_5

<BL_14 EXIT>
    label_14
```

Figure 1. eCFG

The statements in each basic block in Figure 1 exist in tree form. The tree used in this paper is called "*expression tree*" which is used to express each command in the basic blocks as statements in tree form. *Expression tree* is largely divided into an expression derived from the <*Expression*> class which is an abstraction class and a statement derived from the <*Statement*> class which is also an abstraction class. Figure 2 shows a part of BNF that comprises the *expression tree*.

```
Statement → ExpressionStatemnet | 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 → Stmt := PHI(Label = Expression, Label = Expression )
Block → <BL_Label>
Label → label_ Num
Expression → ConstantExpression| DefExpr | StoreExpression |
            ArithExpression
DefExpr → MemExpression
StoreExpression → ( MemExpression := Expression )
MemExpression → VarExpr
VarExpr → LocalExpression | StackExpression
ArithExpression → Expression Op Expression
LocalExpression → (Stack | Local ) Type Num (_undef | _Num)
ConstantExpression → ' ID ' | Num
```

Figure 2. Part of BNF

The derivative classes of <*Expression*> and <*Statement*> are created through the BNF in Figure 2. The difference of these two classes is that the classes derived from <*Expression*> have the *val* field which can maintain values. Furthermore, each <*Expression*> has a *type* field to express the type.

3. DCE

3.1. Setting *DEAD* and *LIVE* to each node

To carry out the DCE process in CTOC, *DEAD* must be set in the *live* field as its initial value which can visit every node in CFG to check dead nodes in preorder. *DEAD* is a *boolean* value which means *false*. This paper uses *DEAD* instead of *false* to clearly show that a node is dead in the *live* field and *LIVE* instead of *true*.

First, all nodes are set to *DEAD*, and before starting the DCE process, the statements and expressions that satisfy specific conditions are reset to *LIVE* in advance, because the statements and expressions preset to *LIVE* are those that always stay live in the program. The statements that must be set to *LIVE* in advance are as follows: First, statements that affect the output of the program; second, assignment statements in which *lhs* corresponding to the *l-value* is used in the live statement; third, conditional statements when one or more statements are activated by control. It is assumed that <*InitStmt*> which means initialization statement in the BNF in figure 2 is always *LIVE*. It is also assumed that <*StoreExpression*> which means assignment expression is *LIVE* if lhs is <*LocalExpression*>. Further, it is assumed that the branched statements such as <*IfStmt*>, <*GotoStmt*>, and <*ReturnExprStmt*> are *LIVE*.

Algorithm 1 is used to describe this process in <*StoreExpression*>.

Algorithm 1. Part of *LIVE* setting algorithm

```
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
end
```

Algorithm 1 checks the statement and expression when a node has *DEAD* as the initial value of the *live* field and sets it to *LIVE* before carrying out DCE. Before setting LIVE, it creates a *worklist* consisting of

connected lists. Then it visits each node and carries out algorithm 1. Algorithm 2 is the node visiting algorithm.

Algorithm 2. Node Visit algorithm.

```
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
end
```

Node visiting is to visit each block and node in the block of CFG in preorder. In figure 1, there are <*InitStmt*>, <*GotoStmt*>, and <*ReturnExprStmt*>. The *live* fields of these nodes are set to *LIVE*. Also, an expression is added to the *worklist* if the current node is <*VarExpr*> which indicates a variable. After Algorithms 1 and 2 are applied to Figure.1, [*Local_ref0_0, Locali2_5*] is set to the *worklist*. *Local_ref0_0* is a variable that means *Dead* which is the class name of the source code, and *Locali2_5* is the output value. These two values are added to the *worklist*, and all the nodes related to these variables are set to *LIVE*.

3.1. Setting *DEAD* and *LIVE* to each node

To remove dead nodes from the tree, all the statements of eCFG are read in the preorder. Then the value of the *live* field of the read node is checked. If the value of the *live* field is *DEAD*, the node must be removed from the tree; otherwise it is in *LIVE* state because it is alive. However, <*LabelStmt*> and <*JumpStmt*> among the *DEAD* nodes are not removed because they represent the information related to branching in the program.

As an example of removing dead nodes, in the block <BL_0> in figure 1, the first statement corresponds to <*LabelStmt*>and its *live* field has the value *DEAD*, but it is not removed because of the reason explained above. The *live* field of the second statement *eval(Locali1_1 := 1)* also has the value *DEAD*. This statement corresponds to <*ExpressionStatement*>. Therefore, it must be removed from the existing statement list. Figure 3 shows the structure of <*ExpressionStatement*>.

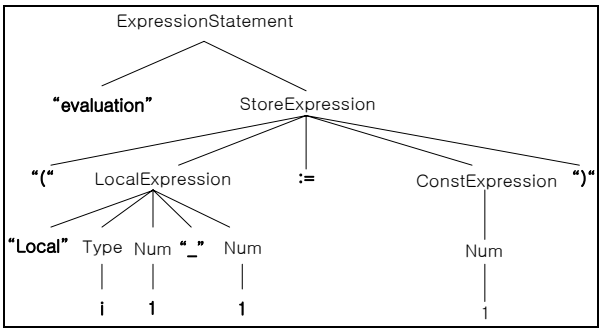

Figure 3. Structure of <*ExpressionStatement*>

In figure 3 if the current node is the root <*ExpressionStatement*>, it is the top node. The parent of this node is the block <BL_0>. To remove this node and the connection between the current node and the parent, the *setParent*() method for parent setting is set to *null*. Moreover, it visits the <*StoreExpression*> which is a child node and removes the connection with its parent node <*ExpressionStatement*>. This can be done by running *setParent*(*null*). In the structure of <*StoreExpression*> in figure 2, there is <*Mem Expression*> on the left of ":=" and <*Expression*> on the right. <*MemExpression*> means the memory, which is the left value (*l-value*) in the assignment statement. In this paper, the left value is indicated by *lhs*. In figure 2 <*VarExpr*> is used for *lhs*. <*VarExpr*> indicates a general variable. In the assignment statement, the right side is the *r-value*, which is represented as *rhs*. *rhs* can be an <*Expression*> such as <*ConstantExpression*>, <*DefExpr*>, <*StoreExpression*>, and <*Arith Expression*>.

Back in the example, first visit *lhs*. In figure 3, *Locali1_1*, which is the *lhs* of <*StoreExpression*> is a <*Local Expression*> derived from <*VarExpr*>. A few things must be considered before removing <*Local Expression*>. If the *lhs* of <*StoreExpression*> is a <*LocalExpression*>, the variables defined in other statements may be used. Therefore, the nodes used in other places must be removed together when the current lhs is removed using the information on the defined variables. In the case of figure 1, when the

definition part of *Locali1_1* in the statement *eval (Locali1_1 := 1)* of <BL_0> is removed, the *Locali1_1* in *eval (Locali1_4 := (Locali1_1 + 3))* was removed as well. Figure 4 shows the final result after carrying out DCE for the **code in Figure 1.**

```
<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
```

Figure 4. Result of DCE.

As shown in figure 4, the statements *eval (Locali1_1 := 1), eval (Locali2_2 := 2)* and *eval (Locali1_4 := (Locali1_1 + 3))* were removed as well because they do not affect the output. However, the statement *eval(Locali2_5 := 4)* is not removed because it is *LIVE* in the previous process and its setting cannot be removed. Furthermore, *label* and *goto* statements are not removed because of the information related to branching.

4. Experiment

The experiment was carried out on a Pentium IV 2.4 GHz PC with 512 MB, using the software Eclipse 3.2 for creating CTOC and testing. For bytecode output, the EditPlus version 2.11 was used. For Java compiler, jdk1.5.0_09 was used.

The example program analyzed seven cases for reviewing control flows to compare the experiment results. The data in the paper by *Don Lance* was used to compare with the experiment results[7]. Table 1 is a simple description of the program used in experiment.

Table 1. Examples of use and simple descriptions.

program	Description
Dead	Example of this paper
SquareRoot	Example of SquareRoot
SumOfSquareRoots	Sum of SquareRoot from 1 to n
Fibonacci	Example of Fibonacci number
BubbleSort	Example of Bubble Sort
LabelExample	Example of labeled break and continue program
Exceptional	try-catch-finally exception handling

Table 2. shows the comparison with the eCFG after conversion to SSA Form.

Table 2. Results after converting to SSA Form.

Program	eCFG lines	SSA lines	%	eCFG 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

In table 2, we can see that the number of lines and nodes generally increased after converting to the SSA From. The reason for this is because statements that did not exist in the existing eCFG increased at the merging point of the control flow. In other words, it was caused by the addition of the Φ-function. However in the case of the *Dead* example used in this paper, no nodes were added because there was no merging point in the program and the insertion of Φ-function did not occur.

Table 3 shows the result after carrying out dead code elimination.

Table 3. Dead code elimination test.

	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

In table 3, while the node count was 31 after the performance of SSA Form, it became 17 after *dead code elimination*. This is because all the nodes of which the live field was *DEAD* were removed during the process. In table 3, no great efficiency of *copy propagation* could be discovered because there was no statements of the form a = b = 1;. Nevertheless, the efficiency of *copy propagation* can be maximized if there are many statements of the form a = b = c = d = 1;. However, we can see that a considerable number of nodes decreased through the dead code elimination. In particular, the *copy propagation* and the *dead code elimination* can be used more effectively by rerunning them after other conversion tasks.

5. Conclusions

As stack-based code, byte code has such shortcomings as slow running speed and being inappropriate for program analysis and optimization. The optimization framework CTOC was constructed for conversion to optimized code. In the optimization process, CTOC used the SSA form which divides variables according to assignment, and the tree structure to express statements.

To carry out the DCE process in CTOC, *DEAD* was set in the *live* field as its initial value which can visit every node in CFG to check dead nodes in preorder. Then, the statements which can set *LIVE* in advance were checked and their live fields were set to *LIVE*, and a *worklist* was maintained for variables that can affect output, and the corresponding variables and related statements were set to *LIVE*. Then, the *dead code elimination* process was carried out for the nodes which stayed *DEAD*. As a result, the nodes that can be removed from the existing nodes could be found, and the optimized tree could be created by removing these nodes.

6. References

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

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

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

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