SSA Form의 효율적 적용

두리단 O 김기태 김재민 유원희

인하대학교

22052190O@inhaian.net, kkt@inha.ac.kr, jeminya@gmail.com, whyoo@inha.ac.kr

An Efficient Application of Static Single Assignment Form

Lidan Du O Kitae Kim Jemin Kim Weonhee Yoo

Inha University

Abstract

Static Single Assignment (SSA) form is an intermediate representation which encodes information about data and control flow that is used to facilitate program analysis and optimization. SSA form simplifies this process with its size linear to program size. Thus we use SSA form to efficiently facilitate bytecode level analysis and optimizations in our CTOC program processor project. In this paper, we illustrate the application and implementation of SSA form using an example. We give the conclusion after experimental results.

1. Introduction

There are some benefits of using a machine-independent intermediate form: retargeting is facilitated; code optimizer can be applied to the intermediate representation [1].

Static single assignment form have been proposed to represent data flow and control flow properties of programs. Variables are names of values. SSA form is a program representation which has such obvious properties:each variable is defined by a single opera-tion in the code, each use of variables refers to a single definition, especially Φ-functions are inserted to merge definitions from different control flow path. The intermediate code is put into SSA form, optimized in various ways, and then translated back out of SSA form. Variants of SSA form have been used for detecting program equivalence and for increasing parallelism in imperative programs. The representation of simple data flow information (def-use chains) may be made more compact through SSA form. Thus, we apply and implement SSA form in our CTOC project in order to improve analysis and SSA form-based optimization of bytecode level program processor.

Java source code is compiled by compiler and class file is generated. Raw bytecode is extracted from class file. Raw bytecode is translated into control flow graph and then SSA form-based CFG. In order to analyze bytecode more efficiently, kinds of optimizations are applied to SSA form-based CFG. At last, optimized SSA form-based CFG is translated back into optimized class file. The input is class file, the output is optimized class file. We name the framework of our project CTOC - Class to Optimized Classes [2]. This paper also contributes an implementation and application of fundamental ideas to yield useful intermediate representations for program analysis.

The rest of this paper is organized as follows. Section 2 gives the related work. The process of translation into SSA form is presented in section 3. Whereas section 4 shows the experimental results. We finish this paper by giving the conclusion.

2. Related work

2.1 Generating control flow graph

Control flow graph (CFG) is a directed graph constructed by set of basic blocks making up a program, to which flow of control information is added. First of all, control flow graph should be pre-constructed. Let us consider a substantial example. In this way, it becomes much easier to illustrate and understand the implementation and application of the algorithms. In figure 1, method Test() written in source code includes two condition statements: one if-then-else statement and one switch-statement. The responding bytecode of this method is transformed by CTOC. Basic blocks are sorted and arranged according to bytecode by translator.

```
public class Example{
     public static int Test(int b){
          int x,y;
          x = 0;
          y = 1;
           if(b < 0)
                  x = -1;
           else {
                 switch(y){
                         case 1:
                              x = 10;
                         break;
                         case 2:
                              x = 20;
                         break;
                         default:
                              x = 30;
                         break;
                 }
           }
           return x;
     }
}
```

Figure 1. Method Test in source code

We only refer to and analyze variable x and ignore variable y and b for space limitation. It is shown that variable x is assigned different values at five different basic blocks or nodes: one is for initialization; the others are for assigning different values to it. X's value is used to turn back the method Test as an integer. Java compiler translates the source code of method Test into raw bytecode. An extended control flow graph should be constructed in advance in order to facilitate the analysis and optimization of bytecode. The transformed CFG is shown in figure 2. BL_42 means block 42. Evaluation statement represents assigning constant or value to variables.

```
<BL_54 HD = null NON In>
     L_54
<BL_55 HD = L_54 NON INIT>
     L_55
          INIT Local_ref0_0 Locali1_1
          goto L_0
     L_53
<BL_0 HD = L_54 NON>
     L_0
          evaluation (Locali2_2 := 0)
     L_2
          evaluation (Locali3_3 := 1)
     L_4
          if0 (Locali1_undef ==0)
               then <BL_13 HD = L_54 NON>
               else <BL_8 HD = L_54 NON>
<BL_8 HD = L_54 NON>
     L_8
          evaluation (Locali2_10 := -1)
          goto L_51
<BL_13 HD = L_54 NON>
     L_13
          switch (Locali3_undef)
               case 1: <BL_36 HD = L_54 NON>
               case 2: <BL_42 HD = L_54 NON>
               default: <BL_48 HD = L_54 NON>
<BL_36 HD = L_54 NON>
     L_36
          evaluation (Locali2_8 := 10)
     L_39
          goto L_51
<BL_42 HD = L_54 NON>
     L_42
          evaluation (Locali2_9 := 20)
     L_45
          goto L_51
<BL_48 HD = L_54 NON>
     L_48
          evaluation (Locali2_6 := 30)
          goto L_51
<BL_51 HD = L_54 NON>
     L_51
          return Locali2_undef
<BL_56 HD = L_54 NON Out>
     L_56
```

Figure 2. Constructed control flow graph

The corresponding graph for figure 2 is shown in figure 3. Condition statement if is located at block 0, branch then is located at block 8 and branch else is located at block 13. Block 0 has two out control flows. Switch statement has three branches: case 1 is located at block 36, case 2 and default branch are located at block 42 and 48 separately.

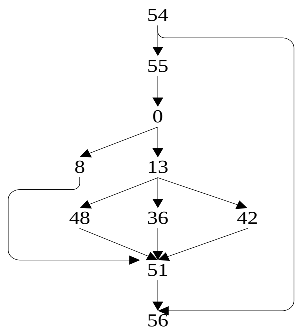

Figure 3. Flow graph

When constructing CFG, block 54 is added as the entry node and block 56 is added as the exit node. In block 0 variable x is firstly initialized by zero and is named as Locali2_2. Control may flow along block 54, 55, 0, 13, 36, 51 to 56, or may flow along block 54, 55, 0, 8, 51 to 56. There are five different flows in figure 3. Variable x is assigned different values -1, 10, 20, and 30; also is given different names. Block 51 is the merging point from block 8, 42, 48 and 36 where x is returned and is named Locali2_undef, which means variable is undefined finally.

2.2 Generating dominator tree

A useful way of presenting dominator information is in a tree, called the **dominator tree**, in which the initial node is the root, and each node d dominates only its descendants in the tree. The existing flow graph is traversed through preorder depth first method. The traversed order starting from zero to the last is block 54, 55, 0, 8, 51, 56, 13, 48, 36 and 42.

The algorithm for building a dominator tree refers to algorithm 1 in [3]. In this algorithm we calculate set of dominators, dom[i]. The high limit of i is the number of blocks in flow graph. For root node dom[0] is initialized by zero; for others (i=1,2,…9), dom[i] is initialized by the whole ten numbers(0,1,2,…9). They are listed in table 1. The value of dom[i] is modified cyclically. In the second cycle, value of dom[4] are changed to 0, 1, 2 and 4, which respond to block 54, 55, 0 and 51. After two cycles, all dom[i] are fixed and not changed any more. So we get final result of dom[i] listed in the last column in table 1.

Table 1. Dominators for each block

	Dom[i]	Initialization	1st	2nd
{54}	dom[0]	0	0	0
{55}	dom[1]	0,1,2,3,4,5,6,7,8,9	0,1	0,1
{0}	dom[2]	0,1,2,3,4,5,6,7,8,9	0,1,2	0,1,2
{8}	dom[3]	0,1,2,3,4,5,6,7,8,9	0,1,2,3	0,1,2,3
{51}	dom[4]	0,1,2,3,4,5,6,7,8,9	0,1,2,3,4	0,1,2,4
{56}	dom[5]	0,1,2,3,4,5,6,7,8,9	0,5	0,5
{13}	dom[6]	0,1,2,3,4,5,6,7,8,9	0,1,2,6	0,1,2,6
{48}	dom[7]	0,1,2,3,4,5,6,7,8,9	0,1,2,6,7	0,1,2,6,7
{36}	dom[8]	0,1,2,3,4,5,6,7,8,9	0,1,2,6,8	0,1,2,6,8
{42}	dom[9]	0,1,2,3,4,5,6,7,8,9	0,1,2,6,9	0,1,2,6,9

Each block or node X has a unique **immediate dominator,** idom(X). Idom(X) should be searched among dom[i]. Idom(X) is the parent of X in dominator tree. For method Test, all idom(X)s are listed in gray color in table 2 at subsection 3.1. The dominator tree for method Test is drawn in figure 4. From this picture, it is easy to figure out that block 0 should be three nodes' dominator: 8, 51 and 13. Block 13 should be the dominator of block 42, 48 and 36, which are three branches in switch or case statement. Meanwhile block 54, 55 and 0 are dominators of the all nodes which follow block 0 in the dominator tree.

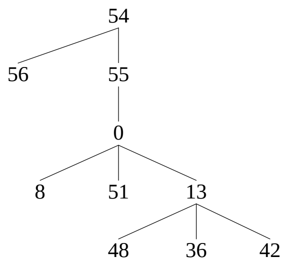

Figure 4. Dominator tree

3. Method for translation into SSA form

The method for translation into SSA form is composed of three sequential steps: first is to calculate dominance frontier for set of nodes in dominator tree; the second is to place Φ-functions into joint statements - the calculated iterated dominance frontier; the last step is to rename the variables in inserted Φ-functions.

3.1 Computing dominance frontier

Given a node X in CFG, the **dominance frontier** of X, DF(X), is a set of nodes Ys, which is the first node (in the series of edges from node X) that is not dominated by node X. To compute the dominance frontier mapping in time linear in the size mapping, two intermediate sets $\mathrm{DF_{local}}(X)$ and $\mathrm{DF_{up}}(Z)$ for each node are defined such that holds the equation (1).

$$DF(X) = \mathrm{DF_{local}}(X) \cup \cup_{z \in \mathrm{Children}\ (X)} \mathrm{DF_{up}}(Z) \qquad (1)$$

Dominance frontier is stored in a linked list. Algorithm 3 in [3] computes dominance frontier for each node X. A bottom-up traversal of the dominator tree is used in this algorithm. The results of dominance frontier for example code can be tabulated in table 2. All successors of their basic blocks should be calculated in advance and listed in the second column. Successor nodes are counted according to control flow graph rather than dominator tree.

Based on the equation (1), the results of $\mathrm{DF_{local}}(X)$ and $\mathrm{DF_{up}}(Z)$ are separately listed in the fourth and fifth column. Algorithm 3 in [3] gives the steps for computing $\mathrm{DF_{local}}(X)$ and $\mathrm{DF_{up}}(Z)$. Children of node X in dominator tree are assigned to node Z. All blanks in the table 2 mean no node or block for that column. In this example, block 55, 0 and 51 have $\mathrm{DF_{up}}(Z)$ only. $\mathrm{DF_{local}}(X)$ for block 8, 13, 42, 48 and 36 are the block 51.

Table 2. Dominance frontier

Block	Succ	Idom	DF_{local}	DF_{up}	DF
Entry{54}	{55,56}				
{55}	{0}	{54}		{56}	{56}
{0}	{8,13}	{55}		{56}	{56}
{8}	{51}	{0}	{51}		{51}
{51}	{56}	{0}		{56}	{56}
Exit{56}		{54}			
{13}	{42,48,36}	{0}	{51}		{51}
{48}	{51}	{13}	{51}		{51}
{36}	{51}	{13}	{51}		{51}
{42}	{51}	{13}	{51}		{51}

We get the dominance frontier for each node in the last column. Block 54 and 56 have no dominance frontier. For block 55, 0, and 51, the calculated dominance frontier includes block 56. As described in the beginning of this section, block 56 is added as an exit node together with entry node 54 in order to construct CFG.

3.2 Placing Φ-functions

Cytron et al.[4] have proved that: the set of nodes that need Φ-functions for any variable V is the **iterated dominance frontier**, $DF^{+}(\varphi)$. We implement the actual computation of $DF^{+}(\varphi)$ by performing as a hash set data structure in algorithm 5 in [3]. The calculated $DF^{+}(\varphi)$ for variable x are block 51 and 56.

We refer to the algorithm 4 in [3] to place Φ-functions. Each real occurrence of variable x is examined. As an example, Evaluation(Locali2_2:=0) in figure 2 is one definition for x. In the algorithm, only non-exit node is the place to be inserted Φ-functions. PhiJoinStmt method defined in other file completes this job.

If other definitions of a variable can be found from other blocks, the variable is set to nonlocal. Algorithm 4 in [3] inserts Φ-functions only for nonlocal variables to implement semi-pruned SSA form which does not perform expensive variable liveness analysis to save time and space.

```
<BL_51 HD = L_54 NON>
    L_51
       Locali2_15 := PHI(L_8=Locali2_undef,
                         L_42=Locali2_undef,
                         L_48=Locali2_undef,
                         L_36=Locali2_undef)
```

Figure 5. Placed Φ-function

In block 51, Locali2_Undef is replaced by Φ-function - Locali2_15 := PHI(). For this example method, there are four different definition points of variable x in CFG. Then four parameters will be added one by one into the parentheses at the right side of the Φ-function. Figure 5 shows the placed Φ-function.

3.3 Renaming variables

Each of the block's successors in the CFG is visited and each of the block's children in the dominator tree is invoked. Block 0, 8, 42, 48 and 36 are the definitions, while block 51 is the use of variable x. We should rename all of them through top down traversal of the dominator tree after the Φ-function is inserted.

When traversing the dominator tree, a stack for variable x is needed to remember the most recently defined version of x. In figure 6, there are four Locali2_undef parameters at the right side of Φ-function in block 48. Among the four Locali2_undef variables, L_8 = Locali2_undef will be firstly renamed as L_8 = Locali2_10. Then the Φ-function is transformed as the following:

```
        Locali2_15 := PHI(L_8=Locali2_10,
                          L_42=Locali2_undef,
                          L_48=Locali2_undef,
                          L_36=Locali2_undef)
```

```
<BL_54 HD = null NON In>
    L_54
<BL_55 HD = L_54 NON INIT>
    L_55
        INIT Local_ref0_0 Locali1_1
        goto L_0
    L_53
<BL_0 HD = L_54 NON>
    L_0
        evaluation (Locali2_2 := 0)
    L_2
        evaluation (Locali3_3 := 1)
    L_4
       if0 (Locali1_1 ==0)
            then <BL_13 HD = L_54 NON>
            else <BL_8 HD = L_54 NON>
<BL_8 HD = L_54 NON>
    L_8
        evaluation (Locali2_10 := -1)
        goto L_51
<BL_13 HD = L_54 NON>
    L_13
        switch (Locali3_3)
            case 1: <BL_36 HD = L_54 NON>
            case 2: <BL_42 HD = L_54 NON>
            default: <BL_48 HD = L_54 NON>
<BL_36 HD = L_54 NON>
    L_36
        evaluation (Locali2_8 := 10)
    L_39
        goto L_51
<BL_42 HD = L_54 NON>
    L_42
        evaluation (Locali2_9 := 20)
    L_45
        goto L_51
<BL_48 HD = L_54 NON>
    L_48
        evaluation (Locali2_6 := 30)
        goto L_51
<BL_51 HD = L_54 NON>
    L_51
        Locali2_15 := PHI(L_8=Locali2_10,
                          L_42=Locali2_9,
                          L_48=Locali2_6,
                          L_36=Locali2_8)
        return Locali2_15
<BL_56 HD = L_54 NON Out>
    L_56
```

Figure 6. Finished SSA form-based CFG

Continually, other three parameters L_36 = Locali2_undef, L_42 = Locali2_undef, and L_48 = Locali2_undef are renamed sequentially as Locali2_8, Locali2_9, Locali2_6. Ultimately, we get the fully Φ-function for variable x as shown in block 51 in figure 6. Also Locali1_undef for variable b and Locali3_undef for variable y are replaced by Locali1_1 and Locali3_3 respectively.

4. Experimental results

Section 3 completes the process of translation into SSA form. We carried out our experiments on Pentium 4 2.0 GHz processor, 512MB Ram, the Java source program and test program are run in Eclipse 3.2.1, Java compiler j2sdk1.4.2_03 is used.

Our experiments include six programs as listed in table 3. CFG lines mean the number of lines in CFG, for example L_0, L_2, L_4 etc. CFG nodes mean numbers of nodes to construct instructions and statements, for example nodes for statement Evaluation (Locali2_2 := 0) etc. The first percentage value column is calculated by the difference between SSA lines and CFG lines divided by SSA lines. The second percentage value column is calculated by the difference between SSA nodes and CFG nodes divided by SSA nodes.

Table 3. Comparison between CFG and SSA

	CFG line	SSA line	%	CFG node	SSA node	%
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
LabelExample	59	63	6.35	58	74	21.62
Exceptional	149	177	15.82	143	304	52.96

The Φ-function is inserted at the merging point in the data flow information in the algorithms for constructing SSA form. After that, both lines and nodes will increase compared to those of original control flow graph. For SquareRoot procedure, the increased percent of lines is 4.76, that of nodes is 15.38.

5. Conclusion

Static single assignment form encodes both control flow and data flow, its features are obvious from the whole process. From control flow graph dominator tree is constructed. Then dominance relation: dominator, immediate dominator, dominance frontier, iterative dominance frontier, is obtained in order to place Φ-functions. The space and time increase because of inserting the so-called Φ-functions. Variables are single in the process of translation into SSA form. SSA form facilitates the process of optimization and analysis of bytecode level. We have efficiently implemented and continue to implement some SSA form-based optimizations in our project.

References

[1] Alfred V. Aho, Ravi Sethi and Jefrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Prentice Hall, New Jersey, 2003.

[2] Young-Kook Kim, Sun-Moon Jo and Weon-Hee Yoo, "Design of Translator for Stack-Based Codes from 3-Address Codes in CTOC ", *ACIS ICIS 2005*, South Korea, pp. 418-423. 2005.

[3] Ki-Tae Kim and Weon-Hee Yoo, "Static Single Assignment Form for Java Bytecodes in CTOC", *The KIPS Transaction: Part D Vol.13-D, No.7,* pp. 939-946, 2006.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM TOPLAS*, *13(4)*, pp. 451-490. 1991.