Optimization in Static Single Assignment Form

External Specification

December 2007
Sassa Laboratory
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

Contents

List of Figures

List of Tables

Chapter 1

Overview

The static single assignment form optimizer (called this system or optimizer hereafter) is a module that performs optimization based on the static single assignment form (SSA form). It is devised as a part of "Research on a common infrastructure for parallelizing compilers" under the Grant "Special Coordination Fund for Promoting Science and Technology" from the Japanese Ministry of Education, Culture, Sports, Science and Technology.

This module is designed for the Low-level Intermediate Representation (LIR) of a common infrastructure for parallelizing compilers called COINS. This module is invoked via a compiler driver, which is also used to invoke various optimizations and analyses involved in COINS. In COINS, any optimization, analysis, or other process invoked via a compiler driver is called "pass". This optimizer is also a pass, which is responsible for the following processing.

- Receiving an LIR from the COINS compiler driver
- Performing optimization on SSA form (*)
- Returning an optimized LIR to the COINS compiler driver

The process (*) is divided into the following part.

1. Translating from normal form into SSA form on an LIR form
2. Performing optimization on SSA form
3. Translating from SSA form into normal form on LIR form

To perform translation from normal form into SSA form, the method developed by Cytron et al., which uses the dominance frontier, has been implemented[11]. The SSA form has several kinds. In this system, three kinds of translation are available via options of the COINS compiler driver: translation into minimal SSA, translation into semi-pruned SSA, and translation into pruned SSA[5]. During translation from normal form into SSA form, copy folding and useless ϕ-function elimination can be performed. This can be performed selectively using an option of the COINS compiler driver. To perform a certain type of loop optimization on a compiler, translating the structure of loops before performing the optimization may provide more efficient translation. However, translating the structure of loops on the SSA form involves rather complicated handling of the ϕ-function. To cope with it, in this optimizer, translating the structure of loops is performed on the normal form, as a process immediately before the process of translating into SSA form. This

can be performed selectively using an option of the COINS compiler driver. For more information on translation from normal form into SSA form, see the Chapter 4.

To perform translation from SSA form into normal form, the method developed by Sreedhar et al. has been implemented[16]. Sreedhar et al. proposed three kinds of algorithms, all of which has been implemented in this system as Method I, Method II, and Method III. Each of these algorithms can be selected using an option for the COINS compiler driver. The system also has a feature of eliminating useless copy statements by doing coalescing during or after the translation of SSA form into normal form. This can be performed selectively using an option for the COINS compiler driver. For more information on translation from SSA form into normal form, see Chapter 5. For optimization on SSA form, the following have been implemented.

- Copy propagation
- Common subexpression elimination
- Global value numbering and partial redundancy elimination based on question propagation
- Constant propagation considering conditional branch
- Dead code elimination
- Elimination of empty blocks
- Elimination of critical edges
- Moving loop invariant expressions out of loops
- Operator strength reduction and test replacement for induction variables
- Removing useless ϕ-instruction
- Algorithms for constructing SSA graph and for finding equality of variables on it

This system allows performing an arbitrary combination of one or more of the above mentioned optimization in an arbitrary order, with each optimizer specified an arbitrary number of times. For more information on optimizations on SSA form, see Chapter 6.

In this system, optimization on SSA form is performed only on virtual registers, to each of which a single value is statically assigned. This is how SSA is implemented in this system. However, in a general C language program, a single value often has two or more aliases such as an array name and a pointed value. The handing of aliases is rather complicated in optimization in SSA form. To cope with it, a method which treats memory locations that are referred to by the program as a single large object, has been implemented in this system as the first approximation of alias analysis. For more information on simple alias analysis, see Chapter 7.

This system treats LIR as not only an input but also an output. It is difficult to verify on LIR whether the optimized code can be executed correctly and whether the LIR translated via optimization is semantically correct. To cope with it, we have implemented a translator from LIR to a corresponding C program. This allows an LIR program to be

translated into a corresponding C program. For more information on translator from LIR to C program, see Chapter 8.

For a list of options for the COINS compiler driver that are applicable to this system, see Appendix A. For general information on COINS, refer to [9].

Chapter 2

Definitions and Preliminaries

2.1 Definitions

The following symbols are used in this specification for explanation. Generally used mathematical symbols are also used.

= This symbol indicates that the left-hand side value (including a variable and a set) equals the right-hand side value (including a variable and a set). For example, a=b means that a equals b. Note that in a C program description or others, this symbol may represent assignment.

← This symbol indicates that a variable or a set on the left-hand side accepts a variable or a set on the right-hand side as an assigned value. For example a←b means that a is assigned to b.

⟶ This symbol indicates that there is a directed edge from the left-hand side entity to the right-hand side entity. For example, a⟶b represents an edge that has a as the initial vertex and b as the terminal vertex.

2.2 Preliminaries

Before going further into optimization on SSA form, this section gives definitions used in this specification of some terms and concepts used for general explanation of compilers and optimizations. For more information on algorithms that are used to calculate the terms and concepts defined here, refer to [1, 13, 3].

2.2.1 Basic Block

A Basic Block is a part (block) of a program, which begins with a single statement starting an execution of the block and ends with an unconditional jump or a conditional jump. Between the beginning and the end, there are no jumps into nor out of the basic block. The control flow graph (described later) that shows basic blocks and the relationship among the basic blocks is widely used as a graphical representation of a program [1, 13, 3]. In Figure 2.1, a C program shown in (a) can be divided into basic blocks as shown in (b). Each rectangle in (b) represents a basic block. For readers' convenience, each basic block has a label such as L1 outside the upper left corner.

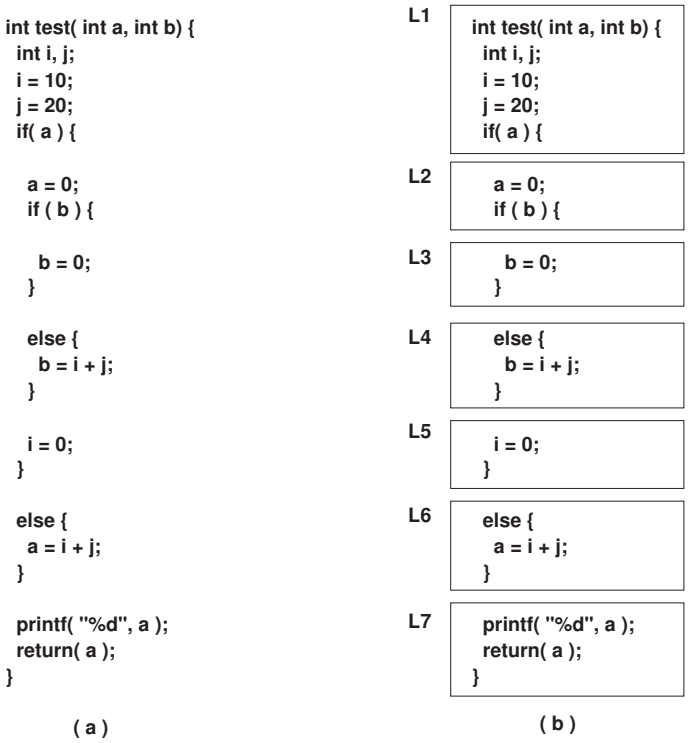


Figure 2.1: Basic block

2.2.2 Control Flow Gaph

A control flow graph (CFG) is a graph that shows a set of basic blocks from the start point of the program to the end point of the program according to the flow of control [1, 13, 3]. Each node of a CFG is a basic block. A directed edge is used to connect nodes. Many methods for optimization in compilers uses information provided by CFG.

Figure 2.2 shows a CFG corresponding to Figure 2.1. Each rectangle in the figure represents a basic block and each arrow represents a directed edge. For readers' convenience, each basic block has a label such as L1 in it. Each node label corresponds to a counterpart in Figure 2.1. Although in some formulation, an entry node and an exit node may be added, the explanation is omitted.

When there are basic block X, basic block Y, and a directed edge from X to Y, X is a predecessor node of Y and Y is a successor node of X. Hereafter, a predecessor node of X is described as $pred(X)$ and a successor node of X as $succ(X)$. For example, the relationship between nodes in Figure 2.2 are described using $pred()$ and $succ()$ as the following.

- L1 : $pred(L1) = \{\}, succ(L1) = \{L2, L6\}$
- L2 : $pred(L2) = \{L1\}, succ(L2) = \{L3, L4\}$
- L3 : $pred(L3) = \{L2\}, succ(L3) = \{L5\}$
- L4 : $pred(L4) = \{L2\}, succ(L4) = \{L5\}$
- L5 : $pred(L5) = \{L3, L4\}, succ(L5) = \{L7\}$
- L6 : $pred(L6) = \{L1\}, succ(L6) = \{L7\}$
- L7 : $pred(L7) = \{L5, L6\}, succ(L7) = \{\}$

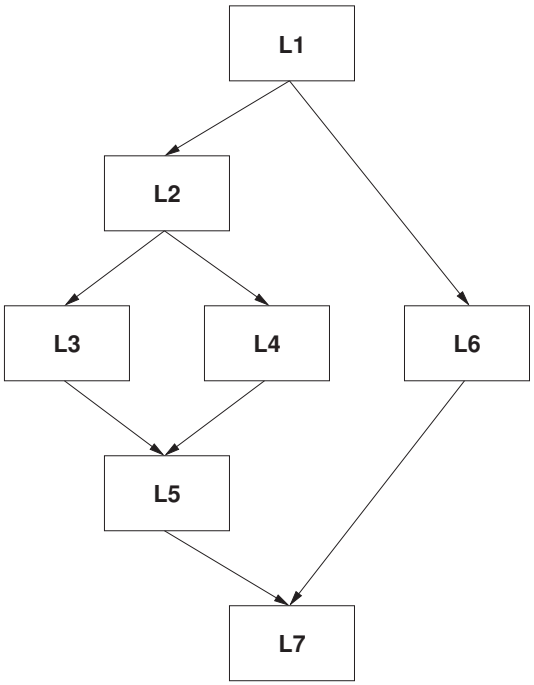


Figure 2.2: Control flow graph (CFG)

2.2.3 Dominance Relation and Postdominance Relation

Dominance Relation

When there are two nodes, X and Y, in a CFG and if any path from the entry of the CFG to Y pass X, we say that X *dominates* Y [1, 13, 3]. In this case, X is a *dominator* node of Y. Hereafter, a set of dominator nodes of X is described as $dom(X)$. The dominance relation is reflexive. Node X dominates node X itself. Also, the dominance relation is transitive. When node X dominates node Y and node Y dominates node Z, node X dominates node Z [13]. For example, in Figure 2.2, the following dominance relations hold.

$$
\begin{aligned}
dom(L1) &= \{L1\} \\
dom(L2) &= \{L1, L2\} \\
dom(L3) &= \{L1, L2, L3\} \\
dom(L4) &= \{L1, L2, L4\} \\
dom(L5) &= \{L1, L2, L5\} \\
dom(L6) &= \{L1, L6\} \\
dom(L7) &= \{L1, L7\}
\end{aligned}
$$

When node X dominates node Y and X≠Y, X *strictly dominates* Y [1, 13, 3]. For example, in Figure 2.2, L1 strictly dominates L3. Hereafter, a set of nodes that strictly dominate X is described as $sdom(X)$.

When node X strictly dominates node Y and there is no node except X that strictly dominates Y on the paths from X to Y, X *immediately dominates* Y [1, 13, 3]. Hereafter,

when X immediately dominates Y, we write $X = idom(Y)$. For example, in the CFG of Figure 2.2, the following immediate dominance relations hold.

$$
\begin{aligned}
idom(L1) &= \{\} \\
idom(L2) &= \{L1\} \\
idom(L3) &= \{L2\} \\
idom(L4) &= \{L2\} \\
idom(L5) &= \{L2\} \\
idom(L6) &= \{L1\} \\
idom(L7) &= \{L1\}
\end{aligned}
$$

Postdominance Relation

When there are two nodes, X and Y, in a CFG and if any path from the exit of the CFG to Y pass X, we say that X *postdominates* Y [1, 13, 3]. In this case, X is a *postdominator* node of Y. Hereafter, a set of postdominator nodes of X is described as $pdom(X)$. The postdominance relation is reflexive. Node X postdominates node X itself. Also, the postdominance relation is transitive. When node X postdominates node Y and node Y postdominates node Z, node X postdominates node Z. For example, in Figure 2.2, the following postdominance relations hold.

$$
\begin{aligned}
pdom(L1) &= \{L1, L7\} \\
pdom(L2) &= \{L2, L5, L7\} \\
pdom(L3) &= \{L3, L5, L7\} \\
pdom(L4) &= \{L4, L5, L7\} \\
pdom(L5) &= \{L5, L7\} \\
pdom(L6) &= \{L6, L7\} \\
pdom(L7) &= \{L7\}
\end{aligned}
$$

When node X postdominates node Y and X≠Y, X *strictly postdominates* Y [1, 13, 3]. For example, in Figure 2.2, L5 strictly postdominates L2. Hereafter, a set of nodes that strictly postdominate X is described as $spdom(X)$.

When node X strictly postdominates node Y and there is no node except X that strictly postdominates Y on the paths from X to Y, X *immediately postdominates* Y [1, 13, 3]. Hereafter, when X immediately postdominates Y, we write $X = ipdom(Y)$. For example, in the CFG of Figure 2.2, the following immediate postdominance relations hold.

$$
\begin{aligned}
pdom(L1) &= \{L7\} \\
ipdom(L2) &= \{L5\} \\
ipdom(L3) &= \{L5\} \\
ipdom(L4) &= \{L5\} \\
ipdom(L5) &= \{L7\} \\
pdom(L6) &= \{L7\} \\
ipdom(L7) &= \{\}
\end{aligned}
$$

2.2.4 Dominator Tree and Postdominator Tree

Dominator Tree

A dominator tree is a graph that is drawn by connecting node X and node Y in a CFG with a directed edge, when X immediately dominates Y. The resulting graph has a tree structure, because for a given node Z in CFG there should be only one node that immediately dominates node Z . The root of a dominator tree is the entry node of the CFG. Hereafter, a child node Y of node X in a dominator tree is described as $Y \in domChild(X)$. Figure 2.3 illustrates the dominator tree corresponding to the CFG in Figure 2.2.

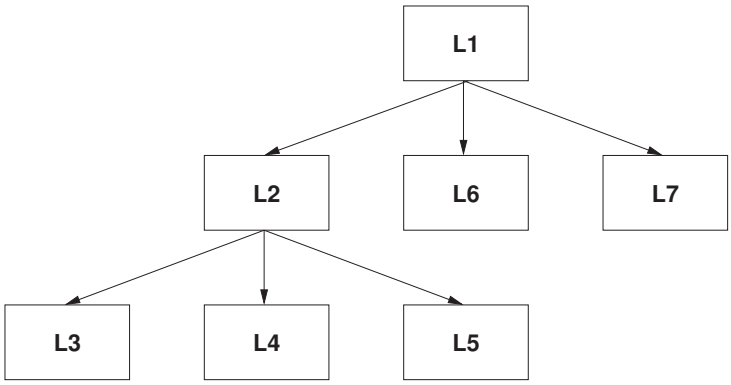


Figure 2.3: Dominator tree

Postdominator Tree

A postdominator tree is a graph that is drawn by connecting node X and node Y in a CFG with a directed edge, when X immediately postdominates Y. The resulting graph has a tree structure, because for a given node Z in CFG there should be only one node that immediately postdominates node Z. The root of a postdominator tree is the exit node of the CFG. Some programs may have no exit node in the corresponding CFG. In this case, the postdominator tree is obtained by adding a virtual directed edge from the entry to the exit of a program. Hereafter, a child node Y of node X in a postdominator tree is described as $Y \in pdomChild(X)$. Figure 2.4 illustrates the postdominator tree corresponding to the CFG in Figure 2.2.

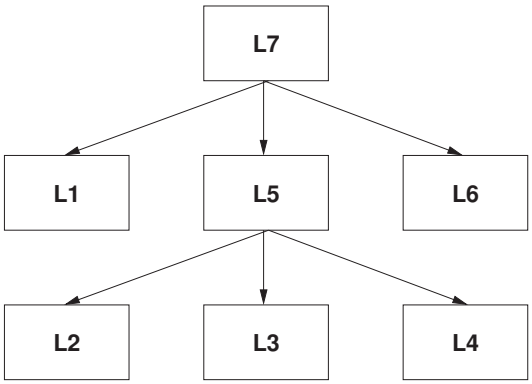


Figure 2.4: Postdominator tree

2.2.5 Dominance Frontier and Postdominance Frontier

Dominance Frontier

Given a node X in a CFG, the dominance frontier of 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. The dominance frontier of node X, $DF(X)$, is defined as the following [13].

$$DF(X) = \{Y | U \in pred(Y)\ exists, X\ dominates\ U, and\ X\ does\ not\ strictly\ dominate\ Y\}$$

In the definition above, the nodes X and U may be the same node. If X and U are the same node, it should be true that $Y \in succ(X)$ and X does not strictly dominate Y. If X≠U, it should be true that $X \in dom(U)$, from the definition. To go further, let us assume that X strictly dominates node Z and consider about $DF(X)$ and $DF(Z)$. If $Y \in DF(X)$ and any path from X to Y includes U, this should result in what Figure 2.5 shows. In this case, the relation $Y \in DF(Z)$ clearly holds.

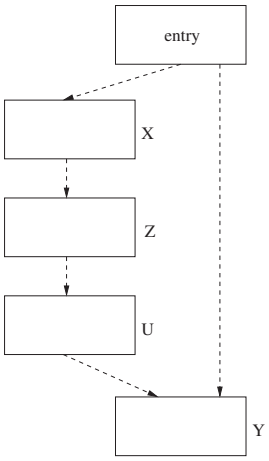


Figure 2.5: Sample case where X's dominance frontier is Y and Z's dominance frontier is also Y

On the contratry, if $Y \in DF(Z)$ but $Y \notin DF(X)$, this should result in what Figure 2.6 shows. Since $Y \in DF(Z)$, a path from the entry node to Y, not including Z, can exist. However, since $Y \notin DF(X)$ is true, any path from the entry node to Y should include X. In other words, in this case, $X \in dom(Y)$ and $X \in sdom(Y)$ hold.

Hence X's dominance frontier is a set of Ys, each of which satisfies the following conditions.

- $Y \in succ(X)$ and $X \neq idom(Y)$.
- When $Z \in dom(X)$, $Y \in DF(Z)$ and $X \neq idom(Y)$.

For example, each node in the CFG shown in Figure 2.2 has a dominance frontier as follows.

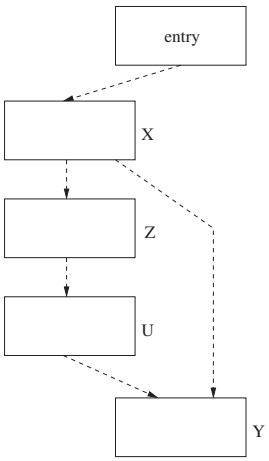


Figure 2.6: Sample case where Z's dominance frontier is Y but X's dominance frontier is not Y

$$
\begin{aligned}
DF(L1) &= \{\} \\
DF(L2) &= \{L7\} \\
DF(L3) &= \{L5\} \\
DF(L4) &= \{L5\} \\
DF(L5) &= \{L7\} \\
DF(L6) &= \{L7\} \\
DF(L7) &= \{\}
\end{aligned}
$$

Postdominance Frontier

Given a node X in a CFG, the postdominance frontier of X is a set of nodes Ys, which is the first node (in the series of edges from node X) that is not postdominated by node X. The postdominance frontier of node X, $PDF(X)$, is defined as the following [13].

$$PDF(X) = \{Y | U \in succ(Y)\, exists, X\, postdominates\, U, X\, does\, not\, strictly\, postdominate\, Y\}$$

Similarly to the dominance frontier, a postdominance frontier is a set of nodes Ys, each of which satisfies the following conditions.

- $Y \in pred(X)$ and $X \neq ipdom(Y)$
- When $Z \in pdom(X)$, $Y \in PDF(Z)$ and $X \neq ipdom(Y)$

For example, each node in the CFG shown in Figure 2.2 has a postdominance frontier, as follows.

$$
\begin{aligned}
PDF(L1) &= \{\} \\
PDF(L2) &= \{L1\} \\
PDF(L3) &= \{L2\} \\
PDF(L4) &= \{L2\} \\
PDF(L5) &= \{L1\} \\
PDF(L6) &= \{L1\} \\
PDF(L7) &= \{\}
\end{aligned}
$$

2.2.6 Natural Loop

One of the most important objects of optimization is a loop. On a CFG, loop structures are divided into two groups; loop structures that can be defined using dominance relations among CFG nodes [1, 3, 13] and loop structures that can not be defined in that way. A natural loop has a loop structure that can be defined using dominance relations among CFG nodes.

Back Edge When a directed edge $b \longrightarrow h$ with b being the start point and h being the end point is given, a directed edge in a CFG whose end point dominates the start point is defined as a *back edge*.

When a back edge $b \longrightarrow h$ is given, a *natural loop* of this directed edge is defined as a union of h and a set of nodes each of which can reach b without going through h. We call h the *header* (entry node) of the loop. Figure 2.7 shows a sample natural loop.

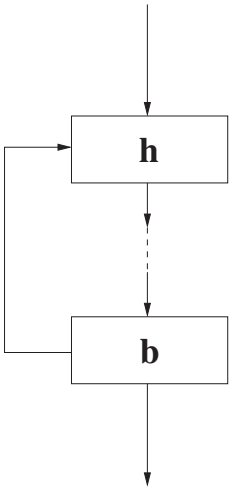


Figure 2.7: Natural loop

A natural loop inherently has the following properties [1].

- A loop must have only one entry node which is called the header. This header node dominates all nodes in the loop. If it did not dominate all nodes in the loop, the header would not be only one entry node of the loop.
- There must be at least one form of repetition in a loop. That is, a loop must have at least one directed edge that returns to the header.

Hereafter a loop that has h as header and a block b from which there is a back edge is described as $LOOP(h, b)$.

2.2.7 Preheader

For many loop optimizations, an instruction sequence (e.g. loop-invariant expression) is inserted into the part preceding the loop execution part. However, suppose that a loop header h can be reached by two or more nodes outside the loop via edges. As mentioned before, an instruction sequence would have to be inserted for each of these nodes. Hence an empty node p called *preheader* is inserted to unify these insertion of instruction sequences. A preheader is inserted outside the loop so that $p \longrightarrow h$ holds. Figure 2.8 shows a sample preheader. Each rectangle in the figure represents a basic block, h is the header node, b is the node with the back edge, and p is the preheader node. In (a) in Figure 2.8, a loop has a header that can be reached by two or more nodes outside the loop via edges. In this case, a preheader p is inserted, as illustrated in (b), to reduce the number of edges from nodes outside the loop to the header to only one.

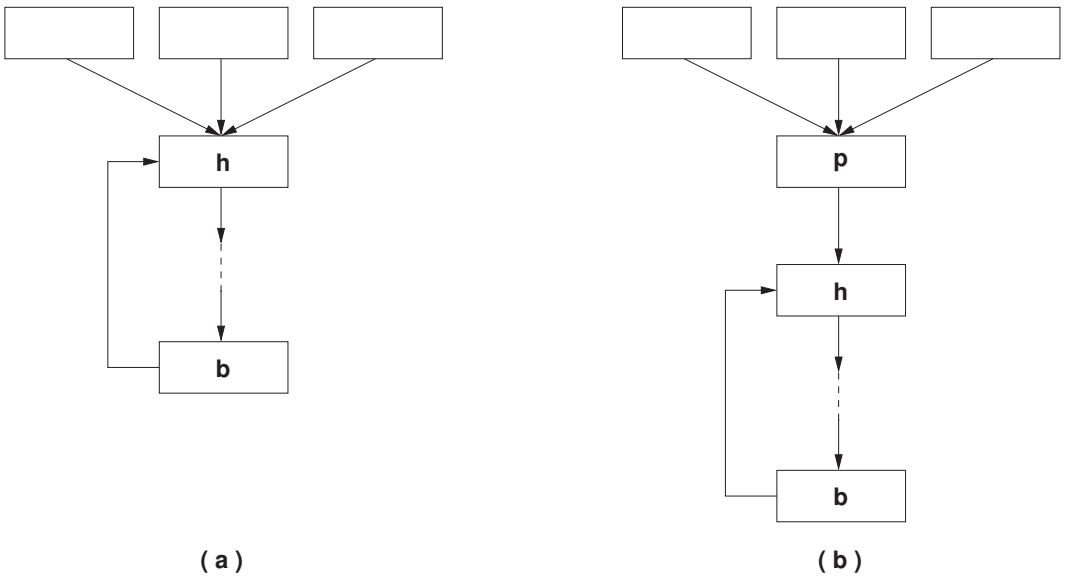


Figure 2.8: Preheader

2.2.8 Control Dependence

When there are two nodes X and Y in a CFG and the following two conditions are satisfied, Y is called to be *control dependent* on X [13].

1. There is a path from X to Y, the path is not empty, and Y postdominates all nodes following X in the path.
2. Y does not strictly postdominate X. That is, $Y \notin spdom(X)$.

Control dependence can be explained in another way. If we follow an edge from X, the path always pass Y. But if we follow another edge from X, the path does not pass Y. Hence the relationship where Y is control dependent on X is equivalent to $X \in PDF(Y)$. Figure 2.9 shows an example control dependence. In this example, Y is control dependent on X.

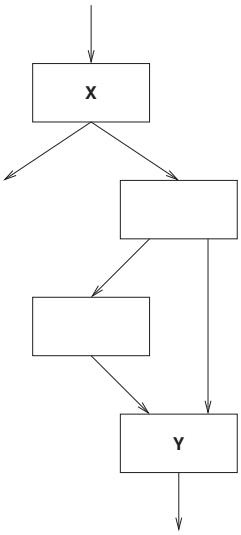


Figure 2.9: Control dependence

2.2.9 Data Dependence

When there is a variable *a* used in a node X of a CFG, any statement in X that precedes the use of *a* does not define *a*, and the variable *a* is defined in a node Y that is on a path from the entry of the CFG to X, X is said to be *data dependent* on Y. In this case, an expression in X that uses the variable *a* cannot be moved beyond Y. Figure 2.10 shows an example data dependence. In this example, X is data dependent on Y.

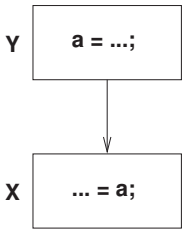


Figure 2.10: Data dependence

Chapter 3

Preprocess before SSA Translation

To provide more effective optimization on SSA form, some cases may require a preprocess where a certain type of translation is executed before optimization. This type of translation may modify the CFG structure by adding nodes to CFG or doing other things. Because the SSA form employs ϕ-functions which are based on CFG structures, it is difficult to modify CFG structures. Hence this system allows several translations to be executed before translation of a program into SSA form.

3.1 Translation of Loop Structure

As for a certain type of loop optimization on a compiler, performing translation of loop structures before the loop optimization may often provide more effective optimization. For example, to safely doing optimazation of moving a loop invariant expressions out of a loop, the block with the loop invariant expression (a node in a CFG) must dominate all exit blocks of the loop to be optimized [1](refer to section 6.4). Under this condition, if we try to move loop invariant expressions out of a loop where the header block and the exit block are the same (i.e. a while style loop), any loop invariant expression that resides in a block of the loop structure other than the header block could not be moved out of the loop. However, by making an equivalence-preserving translation of a while style loop into a do-while style loop will allow any loop invariant expression which resides in a block other than the header block to be moved out of the loop.

In our system, the following have been implemented as translation of loop structures.

- Merging non-nested loops that have a common header [13, 1]
- Translation from while style loop to do-while style loop [13, 1, 3]

A translation of a loop structure modifies the control structure of the program. In the SSA form, the ϕ-function, which is a pseudo code having both of a control flow and a data flow property, is inserted. This implies that modifying control structure of a program would be difficult to handle. Hence translations of loop structures shown below are performed on the normal form immediately before translation to SSA form.

3.1.1 Merging Non-nested Loops that Have a Common Header

According to the definition of the loop presented in section 2.2.6, for two natural loops LOOP(h,b) and LOOP(h',b') with different headers, either the two loops share no common part or one includes the other (that is, they are nested). This is clear from the definition of the natural loop.

However, this definition of natural loops may cause some problem. Loops that have a shared part but do not satisfy the nesting relationship are recognized as different loops. Figure 3.1 shows an example.

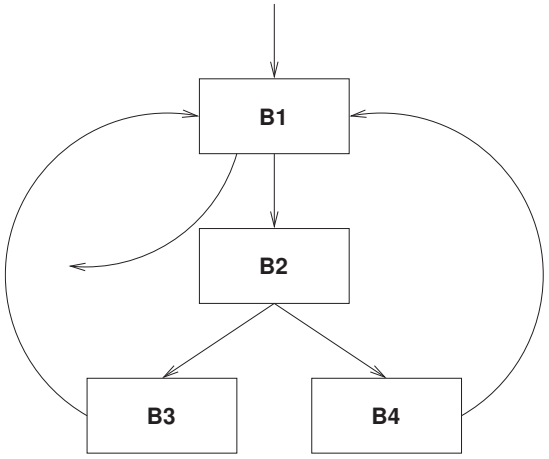


Figure 3.1: Loops sharing a header

In Figure 3.1, the following hold,

$$dom(B3) = \{B1, B2, B3\}, succ(B3) = \{B1\}$$

and the following also hold,

$$dom(B4) = \{B1, B2, B4\}, succ(B4) = \{B1\}$$

hence based on the definition of a natural loop, two loops

$$LOOP(B1, B3), LOOP(B1, B4)$$

can be obtained. However, no nesting relationship exists between these two loops and they share the same header. The resulting two loops are difficult to optimize separately.

To cope with it, these two loops must be integrated into one loop [13, 1]. To apply this translation to the example in Figure 3.1, a new empty basic block Bt is created, back edges $B3 \longrightarrow B1$ and $B4 \longrightarrow B1$ are converted into $B3 \longrightarrow Bt$ and $B4 \longrightarrow Bt$, respectively, and a new back edge $Bt \longrightarrow B1$ is created. The result is shown in Figure 3.2.

As illustrated in this figure, the original two loops LOOP(B1,B3) and LOOP(B1,B4) have been merged into a single loop LOOP(B1,Bt).

3.1.2 Translation from While Style Loop to Do-while Style Loop

The translation from while style loop to do-while style loop is performed by the following steps.

1. Copy the header block H to create block C
2. Insert block C into the back edge of the loop

Figure 3.3 shows a sample program describing a while style loop. Figure 3.4 contains the control flow graph of the program in Figure 3.3. Each rectangle in the control flow

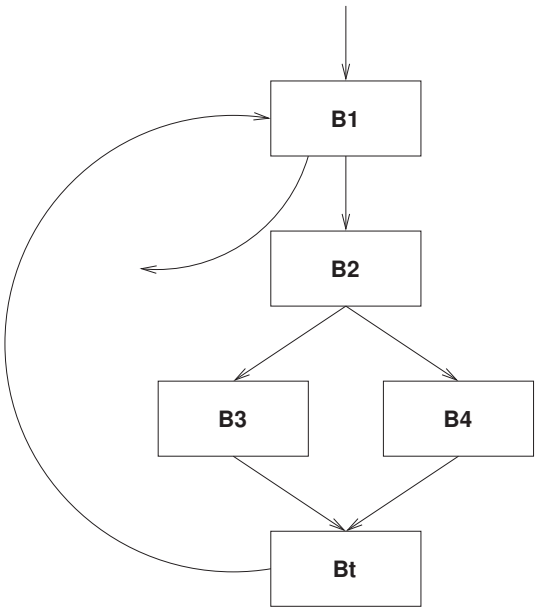


Figure 3.2: A loop resulting from merging loops sharing a header

graph represents a block. Each string in a rectangle represents a pseudo instruction sequence. Each rectangle has a block number label, Li (i represents a numeral), to the left of the rectangle.

In the example in Figure 3.4, blocks L1, L2, and L3 make a loop in which the header is L1 and the exits are L1 and L2. Because this loop has L1, which is the header and an exit, performing loop translation may improve the effect of optimization. Therefore, we copy the instruction sequence in L1 and its successor to create block L1', and we insert it so that the directed edge $L3 \longrightarrow L1$, which is the back edge of the loop, is converted into $L3 \longrightarrow L1'$ (Figure 3.5). Repeating these steps result in translation of a while style loop into a do-while style loop (Figure 3.6).

```
int main(void){
 int i=0,j=0,sum=0;
  int a=10,b=20;
 int x=0,y=0;
LAB001:
 i=i+1;
  j=i+j+1;
 if(i<10 && j<30){
    sum=i+j;
    x=a*b;
   y=a+b;
    goto LAB001;
 }
  i=i+x;
  j=j+y;
 printf("%d,%d\n",i,j);
}
```

Figure 3.3: Sample while style loop

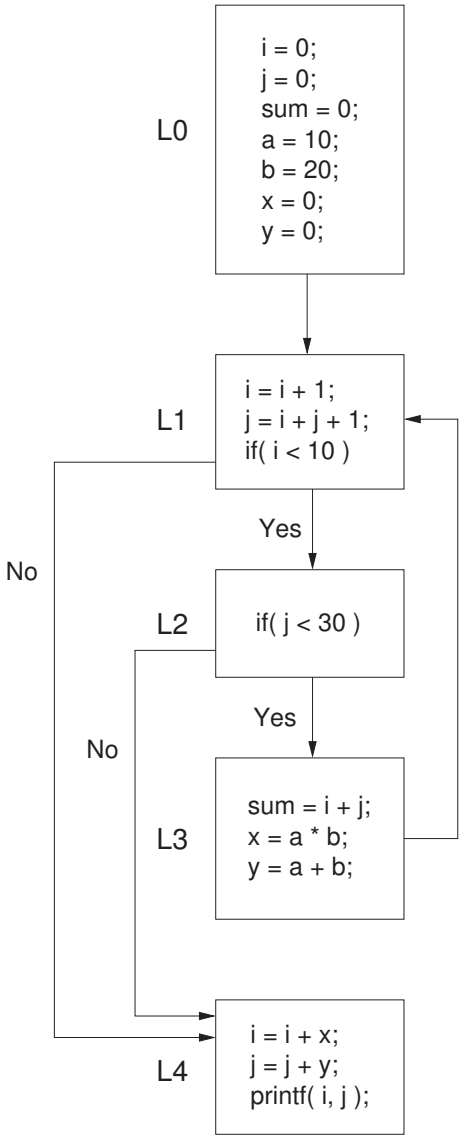
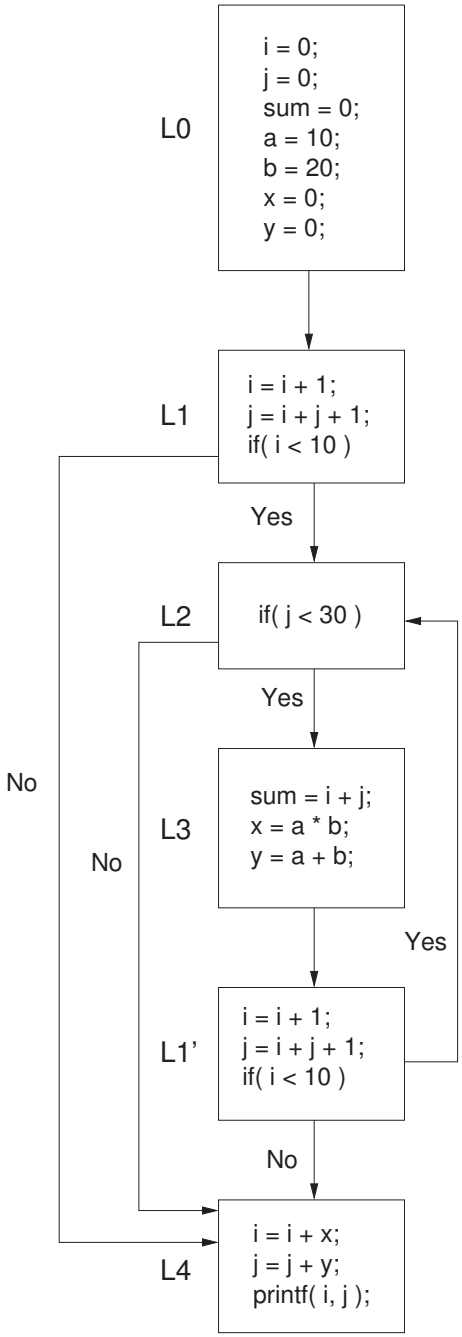
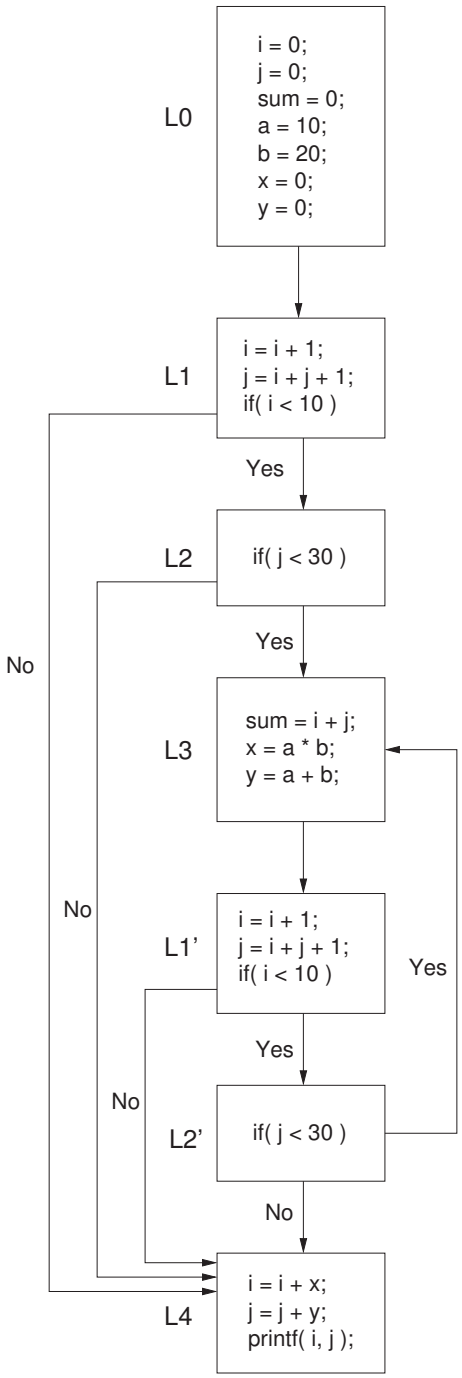


Figure 3.4: Sample while style loop : control flow graph



Block L1' has been added to the CFG in Figure 3.4.

Figure 3.5: Sample while style loop: modified control structure (1)



Block L2' has been added to the CFG in Figure 3.5.

Figure 3.6: Sample while style loop: modified control structure (2)

3.2 Implementation notes

`IF` nodes in LIR have been translated to equivalent code using `JUMPC` nodes before the translation of loop structure which is performed before SSA translation. Thus, there is no `IF` node in SSA form LIR.

Chapter 4

Translation from Normal Form into SSA Form

The algorithm for translation into SSA form implemented in this system is based on the method proposed by Cytron et al., which employs the dominance frontier [11]. Provided that the dominator tree of a CFG is given, the translation algorithm into SSA form implemented in this system is made of the following steps.

1. Obtain the dominance frontier for each node in the CFG.
2. Based on the dominance frontier, obtain nodes to which ϕ-functions are to be inserted for each variable of the normal form. Then insert ϕ-functions.
3. Rename each variable.

In step 2, live range analysis with different precisions can be applied to variables to output three types of SSA forms. Options for the COINS compiler driver are available to specify the type to be output.

4.1 Inserting ϕ-functions

Three variations (types) of SSA forms are widely known: minimal SSA, pruned SSA, and semi-pruned SSA [5, 13]. The difference between the three is achieved during the process of inserting ϕ-functions. The basic algorithm for inserting ϕ-functions is for the minimal SSA form. The pruned SSA and semi-pruned SSA forms can be made by the basic algorithm for minimal SSA form, plus a variation of the live range analysis of variables with a precision specific to the analysis. Figure 4.1 shows the unified algorithm for inserting ϕ-functions. Figure 4.2 illustrates major differences among the three variations of SSA forms.

```
for each block X do
   Initialize Inserted(X) and Work(X)
end for
W ← empty set
for each variable V do
   for each block X with definition of V do
      if translation into semi-pruned SSA is required then
         if V is "non-local" then
            Add X to W
            Work(X) ← V /* This represents that X has been added to W for
            variable V */
         end if
      else /* translation into SSA other than semi-pruned SSA is required */
         Add X to W
         Work(X) ← V /* This represents that X has been added to W for
         variable V */
      end if
   end for
   while W is not empty set do
      Remove block X from W, then for X
      for each Y ∈ DF(X) do
         if Inserted(Y) ≠ V then
            /* φ-functions for V have not yet been inserted into Y */
            if translation into Pruned SSA is required then
               if variable V Lives In into Y then
                  Insert "V ← φ(V, ...., V)" into Y
                  Inserted(Y) ← V /* This indicates that φ-function for V has
                  been inserted into Y*/
               end if
            else /* translation into SSA other than Pruned SSA is required*/
               Insert "V ← φ(V, ...., V)" into Y
               Inserted(Y) ← V /* This indicates that φ-function for V has been
               inserted into Y*/
            end if
            if Work(Y) ≠ V then
               /* Y has not yet been added to W for V */
               Add Y to W
               Work(Y) ← V /* Y has been added to W for V */
            end if
         end if
      end for
   end while
end for
```

Figure 4.1: Algorithm for inserting ϕ-functions

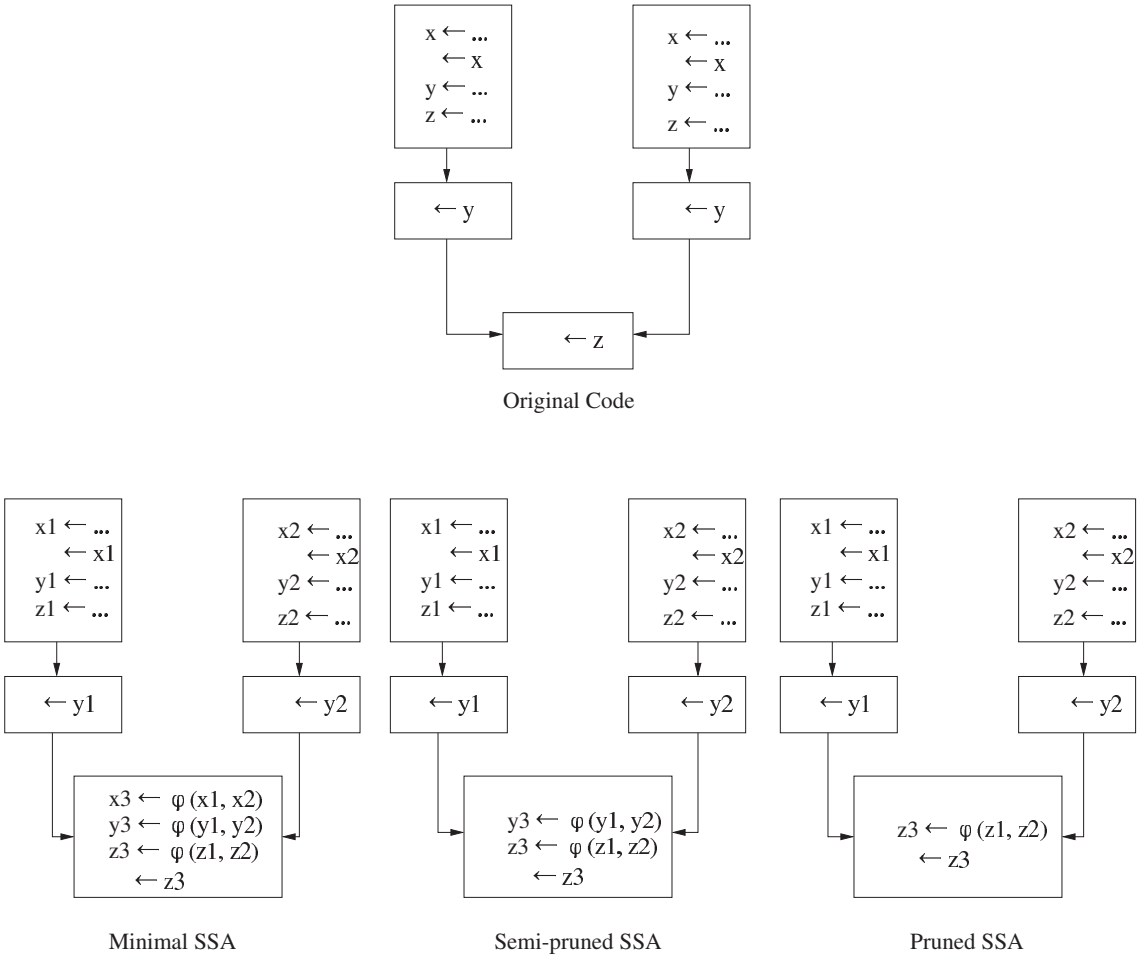


Figure 4.2: Three variations of SSA forms

4.1.1 Minimal SSA

When a definition of a variable V exists in a node X, $Y \in DF(X)$ requires ϕ-functions for V. Defining ϕ-functions for v in Y means adding a definition of v to Y. Therefore DF(Y) also requires to have ϕ-functions for v inserted. Repeat these steps until no dominance frontier is added when ϕ-functions for v have been inserted.

Although minimal SSA is based on a rather simple definition, useless ϕ-functions including ϕ-functions for temporary variables whose live ranges are limited to one basic block may be inserted. Note that "minimal" means that the definition is a "minimal" description, but the greatest number of ϕ-functions will be inserted among the three variations of SSA forms.

4.1.2 Pruned SSA

The minimal SSA form has a disadvantage of having useless ϕ-functions including ϕ-functions for temporary variables as mentioned above. To cope with it, another approach has been developed. Before inserting a ϕ-function, whether or not the ϕ-function is actually needed and useful is verified. Only if the necessity is justified, the ϕ-function is inserted. This approach gives the pruned SSA form.

For pruned SSA, live ranges of variables must be computed in advance. Following the computation the insertion phase arises. Only when a variable v Lives In the basic block that is a dominance frontier of another basic block in which the variable v is defined, the ϕ-function for the variable v is inserted [1]. The case where variable v Lives In the basic block X implies that v is live at the entry of X [2].

Although pruned SSA involves the least number of variables generated during translation into SSA form and is efficient, it needs the cost of live range analysis for every variable, which should be performed in advance .

4.1.3 Semi-pruned SSA

The semi-pruned SSA form is an intermediate between minimal SSA and pruned SSA. The semi-pruned SSA form also uses a specific treatment in inserting ϕ-functions. No ϕ-functions are inserted for the definition of variables that are used only in a single basic block [3]. On the other hand, even if the insertion of a ϕ-function defines a variable that has no other uses afterwards, semi-pruned SSA does insert such a ϕ-function as far as the original variable is used in two or more basic blocks [4].

In semi-pruned SSA, a variable for which a ϕ-function is inserted is called "non-local" [6, 5]. The semi-pruned SSA form requires any "non-local" variable to be found in advance. However, this approach takes lower cost than the approach of analyzing the live range of every variable. To find "non-local" variables, only a single scan of the whole basic blocks is required. When a variable v used in a basic block has not been defined before this use in the current basic block, the variable v is recognized as a "non-local" variable.

[1] z in Figure 4.2represents the inserted function
[2] A case where a variable v lives out a basic block X implies that v lives at the exit of X
[3] x in Figure 4.2 represents this application
[4] y in Figure 4.2 represents this application

4.2 Renaming of Variables

4.2.1 Algorithm

The SSA form requires that variables that have the same name have the same value. In other words, whenever a variable is defined, a unique name must be given to the variable. Renaming of variables is performed recursively, traversing the dominator tree depth-first from the root. Figure 4.3 shows the algorithm used for renaming of variables [13, 3].

The SSA form assumes that all variables have been initialized in the entry block of the CFG. In our algorithm, such an initial value is described as $\perp$.

In this algorithm, a piece of code assigning an initial value is inserted at the beginning of the program for each such $\perp$. As the initial value, 0 is assigned to variables of the integer type (the type represented as Ixx in LIR) and 0.0 is assigned to variables of the floating-point type (the type represented as Fxx in LIR).

4.2.2 Copy Folding

This system allows Copy Folding to be performed simultaneously with translation of a program into SSA form. Copy Folding is a method used for deleting a copy statement such as a←b in source code. The function of Copy Folding is the same as that of copy propagation to be presented in Section 6.1. In our system, the term "Copy Folding" is used to represent the copy propagation that is performed during translation into SSA form [5].

The Copy Folding method takes notice of copy statements during the phase of renaming variables in the translation into SSA form. When the statement to be renamed is a copy statement a←b, b is put onto the stack of the destination variable of the copy statement, Stack(a), and the statement A is deleted. Hence any use of a in the following expression is renamed to b.

For more information, refer to the algorithm for renaming of variables (Figure 4.3).

```
for each variable v do
   /**
     * Initialize the stack into which variables will be put.
     * The algorithm introduced in the first edition of
     * "Structure and Optimization of Compilers by I. Nakata"
     * lacks the process of assigning initial values.
     **/
   Stack(v) ← ⊥
end for
call renameVariables(Entry)

renameVariables(X) /* Procedure renameVariables() */
  for each statement A in block X do
  /* from the beginning of the block in a sequential manner*/
     if the right-hand side of statement A is not a φ-function then
     /* this includes conditional statements */
       for each variable v in the right-hand side of statement A do
       /* this includes variables occurring in a conditional expression */
         Replace v in the right-hand side with the top of Stack(v)
       end for
     end if
     for each variable v in the left-hand side of statement A do
     /* this includes the case where the right-hand side has a φ-function */
       if statement A is a copy statement and Copy Folding is specified then
         Put variable vr in the right-hand side onto Stack(v)
         Delete statement A
       else
         Put a new variable v_new onto Stack(v)
         Replace v with v_new
       end if
     end for
   end for
   for each block Y ∈ succ(X) do
     Replace parameter v of φ-function in Y with v_new
   end for
   for each block Z ∈ domChild(X) do
     call renameVariables(Z)
     /* rename along the dominator tree from the top to the bottom*/
   end for
   Pop all the variables that have been put onto the stack during the processing of
   block X
```

Figure 4.3: Algorithm for renaming of variables

4.3 Initial values of SSA form variables

In the current implementation, values 0 or 0.0 are assigned as initial values of SSA form variables instead of $\perp$ (section 4.2.1). To handle optimizations such as constant propagation and others more precisely, it is desirable to use $\perp$ as intitial values.

4.4 Restriction of the implementation

`PARALLEL` and `SUBREG` which are specified in the documentation of LIR have some problems when we use them in SSA form. If we have to treat `PARALLEL` strictly, we can do it by breaking down `PARRALEL` considering the simultaneous assignment property and then translating it to SSA form. However, this process invalidates the effect of other optimization using `PARALLEL`, such as SIMD optimization. In addition, `SUBREG` means that assignment of a value to a part of one register variable is made, and it is difficult to express this correctly in SSA form. Therefore, our system does not optimize the L function where such L expressions reside.

The management of `IF` node in LIR is described in section 3.2.

Chapter 5

Translation from SSA Form into Normal Form

This chapter describes the basic method and its problems in the translation from SSA form into normal form, then, describes the algorithm of Briggs et al. [5] and the algorithm of Sreedhar et al.[16] which we implemented.

ϕ-functions included in the SSA form are hypothetical funtions to satisfy the definition of SSA form, and cannot run on the ordinary computer. Therefore, elimination of the ϕ-functions from the program in SSA form is necessary. It has been pointed out that the translation has some critical problems, so we also have to consider about that.

This system basically use the algorithm of Sreedhar et al. We implemented it at first. Then we inserted the implementation of the algorithm of Briggs et al. by Kohama [12] into it. We recommend to use Method III by Sreedhar et al.

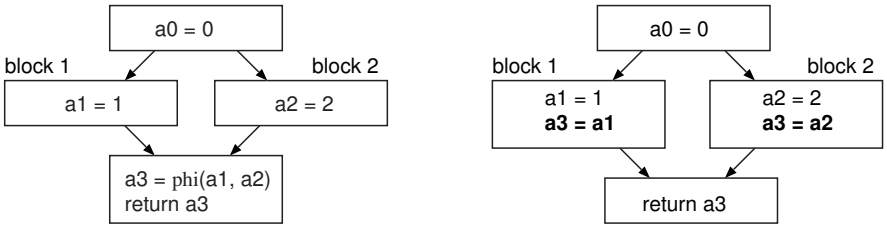
5.1 Outline of translation from SSA form into normal form

First, we explain the naive back translation algorithm which Cytron et al.[11] proposed. In the control flow graph (CFG) at the left of Figure 5.1, if the control comes from `block1`, the value of `a3` becomes `a1`, and if the control comes from `block2`, the value of `a3` becomes `a2`. Thus, the ϕ-function can be eliminated by inserting copy statements "`a3` $\leftarrow$ `a1`" at the end of `block1` and "`a3` $\leftarrow$ `a2`" at the end of `block2`, as in the CFG at the right of this figure. In general, copy statements are inserted at the end of every control flow predecessor of the basic block where the ϕ-function resides. The ϕ-functions are replaced by these copy statements and the ϕ-functions are removed. We call this algorithm as the naive back translation algorithm.

5.2 Critical problems at translation from SSA form into normal form

It has been pointed out that the translation from SSA form into normal form with the naive back translation algorithm has some critical problems. We show these problems with the examples and explain the reasons. In the following, we call the variables which appear in the left-hand side of the ϕ-functions and copy statements as **target**s, and the variables which appear at their right-hand side as **source**s.

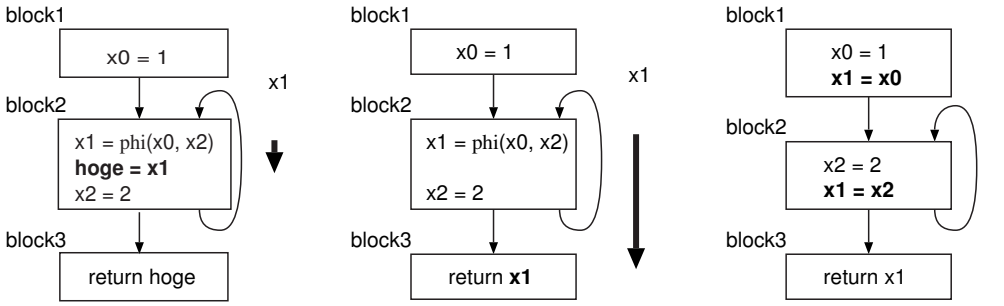
As the live range of the variables in the ϕ funtion is focused in the following discussion, we explain how to handle it.



SSA form

After back translation from SSA form

Figure 5.1: Naive back translation algorithm



Graph1:
SSA form

Graph2:
After copy propagation

Graph3:
After back translation
using the naive method

Figure 5.2: Extension of live range

- The live range of the target in the ϕ-function must include the entry of the block where the ϕ-function resides, but it does not include the exit of the predecessors of the block.
- The live range of the source in the ϕ-function must include the exit of the predecessors of the block containing the ϕ-function, but does not include the entry of the block where the ϕ function resides.

5.2.1 Problem related to extension of the live range (lost copy problem)

A variable in the program is called **live** at a point, if it is used later in the program. On the other hand, a variable in the program is called **dead**, if it is not used later in the program, When the variable is checked in the reverse order of the execution of the program, the span from "the first appearance of its use" to "its definition" is the range where the variable is alive, and is called **live range**.

First, consider the program of Graph 1 in Figure 5.2. The arrow beside the graph describes the live range of `x1` which is the target of the ϕ-function. After applying copy propagation optimization to the Graph 1, copy statement "`hoge` $\leftarrow$ `x1`" is removed, and "`return hoge`" in the `block3` is replaced by "`return x1`"(Graph 2 of Figure 5.2). As described in Graph 2, the usage of `x1` extends forward, and the live range of `x1` goes beyond `block2`.

Here, consider the translation of Graph 2 in Figure 5.2 to normal form. According

to the naive back translation algorithm (section 5.1), copy statements are inserted at the end of the predecessors of the block where the ϕ-function resides. In this case, "`x1` $\leftarrow$ `x0`" is inserted to `block1`, and "`x1` $\leftarrow$ `x2`" is inserted to `block2` (Graph 3 in Figure 5.2). However, the live range of `x1` includes the exit of `block2` in Graph 2. If "`x1` $\leftarrow$ `x2`" is inserted at the end of `block2`, the value of `x1` is rewritten incorrectly in the live range of `x1`, and the value of `x1` which is used at `block3` might be destroyed. Actually, if the control goes through `block1-block2-block3`, the `return` value at `block3` is 1 in Graph 1 and Graph 2, but the `return` value at `block3` is 2 in Graph 3, so the meaning of the program is changed. This problem is called the **lost copy problem**.

Thus, when translation from SSA form into normal form is performed, if the target of the copy statement to be inserted is live at the point where the copy statement should be inserted, it is necesary to solve this problem.

5.2.2 Problem related to simultaneous assignment property of ϕ-functions (simple ordering problem)

If there are plural ϕ-functions in the same block, these ϕ-functions must be regarded to be processed simultaneously. It is called **the simultaneous assignment property of the ϕ-function**.

Now, consider the program represented by Graph 1 in Figure 5.3. If copy propagation is applied to Graph 1, then "`y2` $\leftarrow$ `x1`" in `block2` is deleted, and "`y1` $\leftarrow$ ϕ`(y0, y2)`" is replaced by "`y1` $\leftarrow$ ϕ`(y0, x1)`" (Graph 2 of Figure 5.3). In such case, two ϕ-functions in the same block, "`x1` $\leftarrow$ ϕ`(x0 , x2)`" and "`y1` $\leftarrow$ ϕ`(y0 , x1)`", should be processed simultaneously. If the control goes through `block1-block2-block2-block3`, the value of `x1` which is assigned to `y1` in `block2` at the second time is not the value of `x1` assigned in `block2` at the second time, but, it is the value assigned in `block2` at the first time.

Here, consider the translation of Graph 2 in Figure 5.3 into normal form. In the naive back translation algorithm, we insert

```
x1 ← x0
y1 ← y0
```

or,

```
y1 ← y0
x1 ← x0
```

into the end of `block1`. In a similar way, we insert

```
x1 ← x2
y1 ← x1
```

or,

```
y1 ← x1
x1 ← x2
```

into the end of `block2`.

The ϕ-functions should be processed as having the simultaneous assignment property, hence, the order of the copy statements to be inserted is supposed to be irrelevant. However, if we see the set of copy statements to be inserted into `block2`, the value that is copied to `y1` might be different whether the former order or the latter order of copy statements

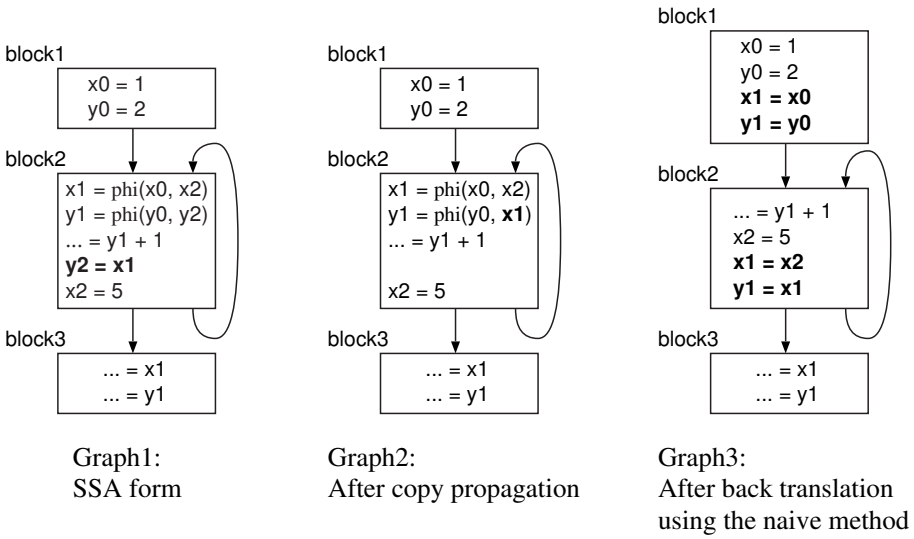


Figure 5.3: Simultaneous assignment property of ϕ-functions

is used (Graph 3 of Figure 5.3). In Graph 3, the values of `x1` and `y1` in the `block3` always become `5`, which is different from Graph 1. This problem is called the **simple ordering problem**.

Thus, if there are a number of ϕ-functions in the same block, it is important to pay attention to the simultaneous assignment property of the ϕ-functions, when translation to the normal form should be performed.

5.3 Method of Briggs et al.

The SSA back translation algorithm by Briggs et al.[5] is an algorithm which solves the problem pointed at section 5.2. This section explains the algorithm by Briggs et al.

5.3.1 Principle

The principle of the algorithm of Briggs et al. is that appropriate copy statements are inserted at the end of every control flow predecessor of the basic block where the ϕ-function resides, as in the case of the naive back translation algorithm. Namely, it is a method that the copy statements substitute ϕ-functions. When critical situation occurs, the problem is avoided by saving the values or scheduling the copy statements.

If we adopt this principle, a lot of copy statements are inserted to the program. However, Briggs et al. claim that coalescing the live ranges at the register allocation phase after the back translation can eliminate many of these copies.

5.3.2 Solution of the lost copy problem

In section 5.2.1, we explained the crisis that the expected target value might be destroyed by insertion of the copy statement when it is inserted into the live range of the target of the ϕ-function, in translating SSA form into normal form (lost copy problem). To avoid such crisis, Briggs' method saves the value of the target into another variable, and keeps the information that the value of the target is saved in that variable, then replaces the use of the target in later blocks by the variable which saves the value. In this way, Briggs' method can eliminate the influence on later blocks, even if the value of the target is destroyed by copy statements at the end of the block.

In the algorithm, we assign the value of the target which has not yet been destroyed to `temp`, by inserting the copy statement such as "`temp` $\leftarrow$ `target`" into the beginning of the block where the ϕ-function resides. At the same time, we prepare the `Stack` which has variable names as its index, and save the information that the value of the target is saved in `temp`. To be more precise, first, push `temp` on to `Stack[target]`, then replace the targets used in later blocks by `temp` which is in `Stack[target]`.

For example, consider the upper graph in Figure 5.4. [1] In the naive back translation algorithm, the copy statement "`x1` $\leftarrow$ `x2`" is inserted into the end of the predecessor block `block2`, but the inserted point is in the live range of `x1`, so "`temp` $\leftarrow$ `x1`" is inserted into the beginning of `block2` and the value of `x1` is stored into the new variable `temp`. Next, `temp` is pushed on `Stack[x1]`, and the information that the prior value of `x1` before rewriting by copy statement is stored at `temp` is memorized.

After that, `x1` used in `block3` is replaced by `temp` which was pushded on the `Stack[x1]` (lower graph of Figure 5.4).

With this process, ϕ-functions can be removed without changing the meaning of the program, even if "`x1` $\leftarrow$ `x2`" is inserted (Figure 5.5).

[1] We draw only the live range in the block `block2` which we should take notice with an arrow, and if it includes the entry or exit of the block, we specify it with $\bullet$. In addition, the part of the program which is written in bold-face means the part which is going to be inserted, and is not reflected to the live range. In the following part of this section, we use this convention. In addition, Figures describe the way of thinking, and do not correspond to the algorithm one by one.

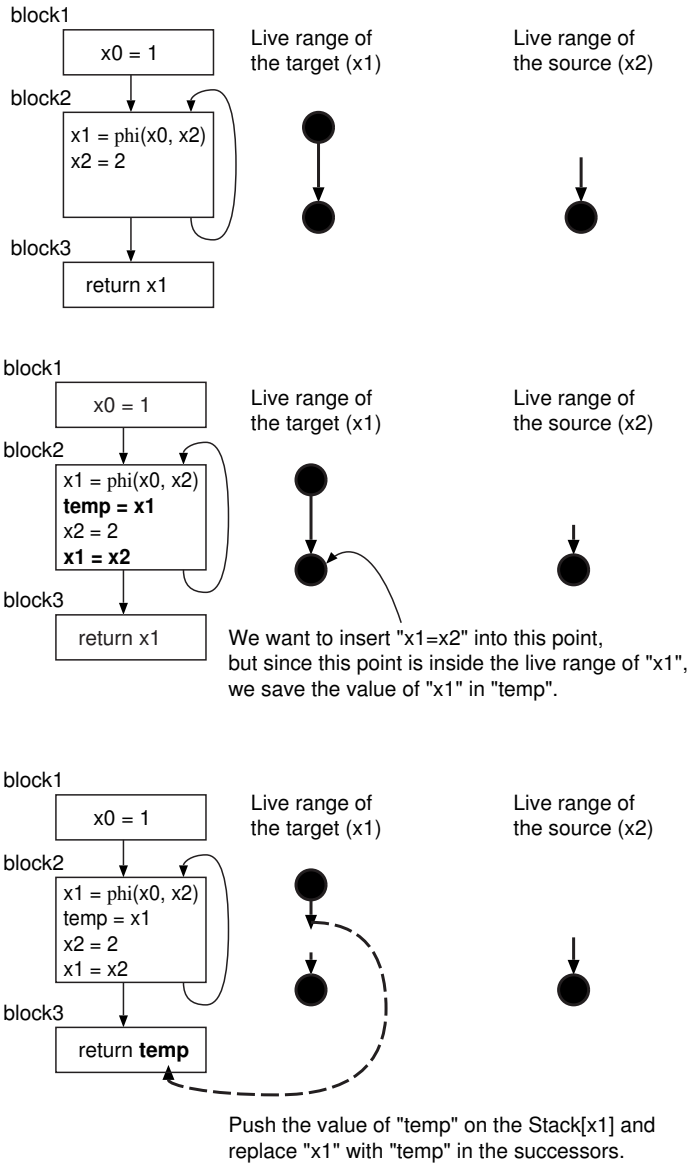


Figure 5.4: Translation process of the lost copy problem

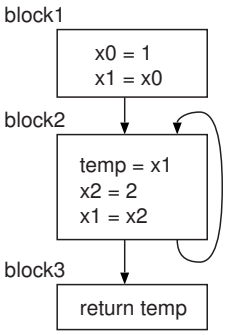


Figure 5.5: Result of translation for the lost copy problem

	target	source
x1 ← x2	x1	x2
y1 ← x1	y1	x1

Table 5.1: Relationship between copy statements and ϕ-functions

5.3.3 Solution of the simple ordering problem

In section 5.2.2, we described the crisis that if plural ϕ-functions are in the same block but the simultaneous assignment property of the ϕ-functions is not considered, the expected value might be destroyed (the simple ordering problem). To avoid this problem, in Briggs' method, when inserting copy statements, the copy statements are scheduled so as not to change the meaning of the program.

For example, see the upper graph of Figure 5.6. In the naive back translation algorithm, copy statements, "x1 ← x2" and "y1 ← x1" are inserted into the end of the predecessor block block2, but the value of y1 may be different depending on the order of the inserted copy statements.

Table 5.1 helps to understand the simultaneous assignment property of the ϕ-functions. It describes the relationship between the two copy statements described above and the target and source of the corresponding ϕ-functions.

In this table, the variables in the item of target are the variables whose value is rewritten when this copy statement is executed, and the variables in the item of source are the variables whose value should not have been rewritten when this copy statement is executed. Now, consider x1. x1 is the variable which is rewritten when "x1 ← x2" is executed, and also the variable which should not have been rewritten when "y1 ← x1" is executed. It means that if the program is scheduled as to execute "y1 ← x1" first and then "x1 ← x2" next, the expected value is set to variable y1 when "y1 ← x1" is executed, because x1 has not yet been rewritten.

Then, we discuss Figure 5.6 again. In Briggs' method, the target of each ϕ-function in the block is first checked wheather it is not the source of another ϕ-function in the same block. In this example, it is found that x1 which is the target of "x1 ← ϕ(x0, x2)" is also the source of "y1 ← ϕ(y0, x1)". So the first step is the insertion of "y1 ← x1" which should be executed first. However, because the inserting point is inside the live range of y1, we insert "temp ← y1" into block2 to save the value of y1 into temp, and then insert "y1 ← x1" (the upper graph of Figure 5.6).

The next step is the insertion of "x1 ← x2". Since the inserting point is inside the live range of x1, we insert "temp' ← x1" into the top of block2 to save the value of x1 into temp', and then insert "x1 ← x2" (the middle graph of Figure 5.6).

Finally, x1 and y1 which are used in block3 are replaced respectively by temp' and temp, which are pushed on Stack[x1] and Stack[y1] respectively (the lower graph of Figure 5.6).

Through these steps, ϕ-functions can be removed without changing the meaning of the program (Figure 5.7).

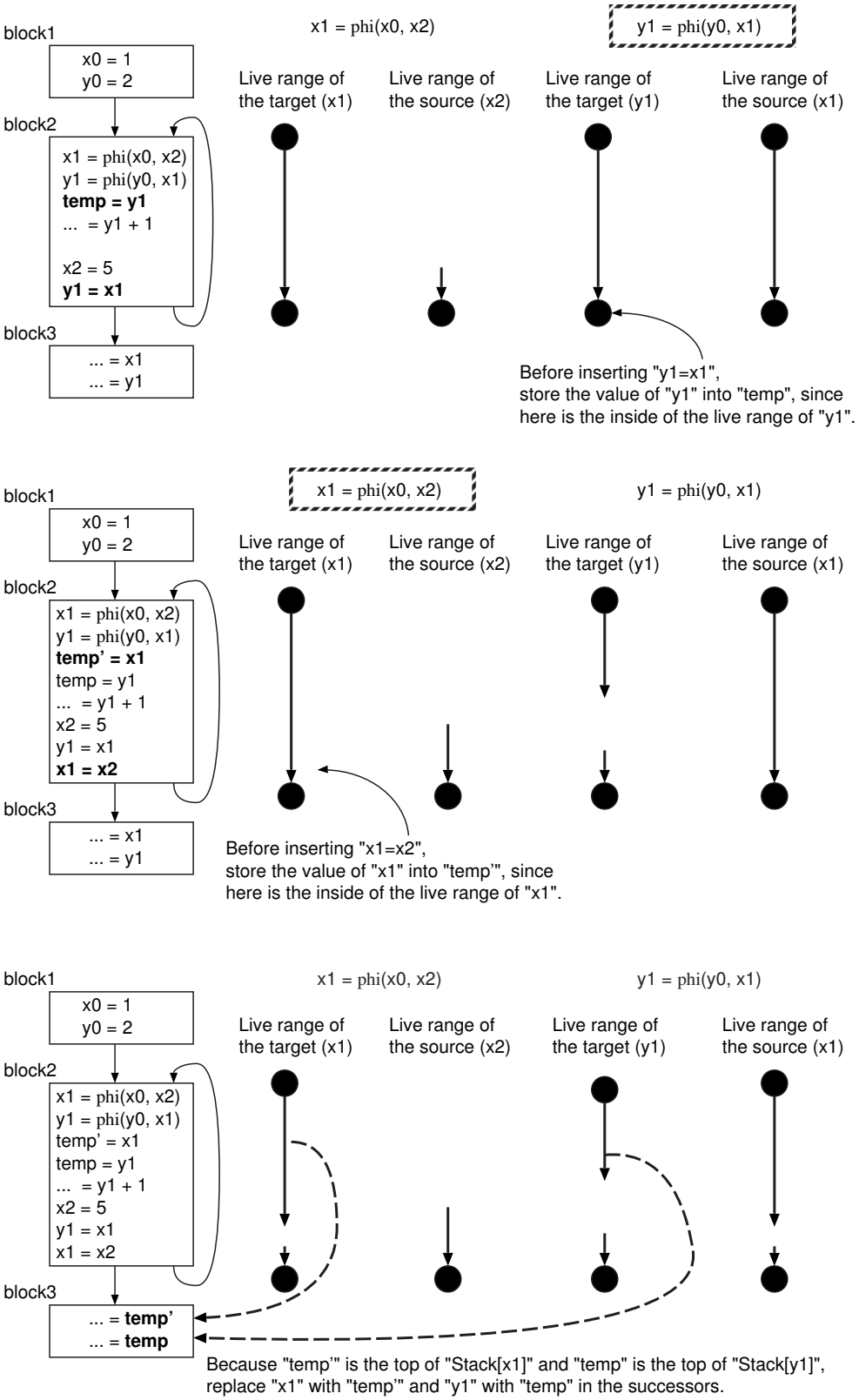


Figure 5.6: Translation process for the simple ordering problem

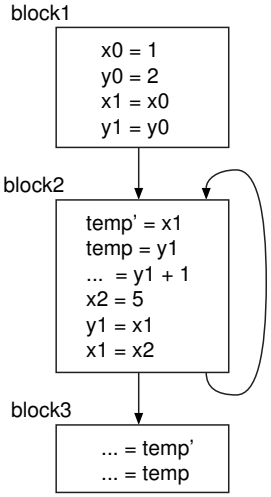


Figure 5.7: Result of translation for the simple ordering problem

The swap problem

Thus far, it becomes clear that when the target of a ϕ-function is the source of another ϕ-function, the problem can be avoided by insertion of the copy statements in the proper order, However, actually the targets of two ϕ-functions might be the sources of each other. Namely, the following case is possible.

```
x1 ← φ(x0, y1)
y1 ← φ(y0, x1)
```

This problem is called the **swap problem**(Figure 5.8).

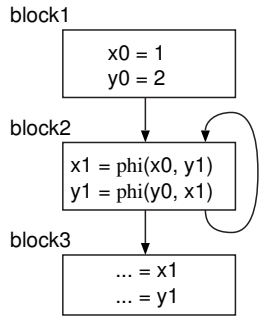


Figure 5.8: The swap problem

In the naive back translation algorithm, "`x1 ← y1`" and "`y1 ← x1`" are inserted into `block2`. In this case, the result is not the expected one whichever copy statement is inserted first. To avoid this problem, choose the copy statement corresponding to one of the ϕ-functions, and save its target to another variable temporarily. For example, if "`x1 ← y1`" is chosen, save the value of `x1` into `temp` before execution of "`x1 ← y1`", as described below,

```
temp ← x1
x1 ← y1
y1 ← temp
```

then the proper value is assigned to `y1`.

Now we explain a concrete translation process. If we choose the copy statement "`x1` ← `y1`" corresponding to "`x1` ← ϕ`(x0, y1)`" in the upper graph in Figure 5.9, first insert "`temp` ← `x1`" into the end of `block2`. At this moment, temporarily keep the information that the value of `x1` is saved to the variable `temp` as `map[x1]` ← `temp` (the upper part of Figure 5.9). The difference between `Stack` and `map` is that `map` is used only when the value of `x1` is needed at the insertion of the copy statement in the same block.

Next, insert the copy statement "`x1` ← `y1`". Because the inserting point is in the live range of `x1`, we need to insert "`temp'` ← `x1`" and push `temp'` on `Stack[x1]`. Then, insert the copy statement "`y1` ← `x1`" corresponding to "`y1` ← ϕ`(y0, x1)`". At this moment the information of `map` is used. `map[x1]`← `temp` indicates that the value of `x1` has been kept in `temp` temporarily, so "`y1` ← `temp`" is the copy statement to be inserted. Since the point where the copy statement is to be inserted is also in the live range of `y1`, we process it similarly to the above (the middle part of Figure 5.9).

Finally, replace `x1`, `y1` in `block3` by `temp'`, `temp''` which have been pushed on `Stack[x1]`, `Stack[y1]` respectively (the lower part of Figure 5.9).

Thus, the ϕ-functions can be removed without changing the meaning of the program by these processes (Figure 5.10).

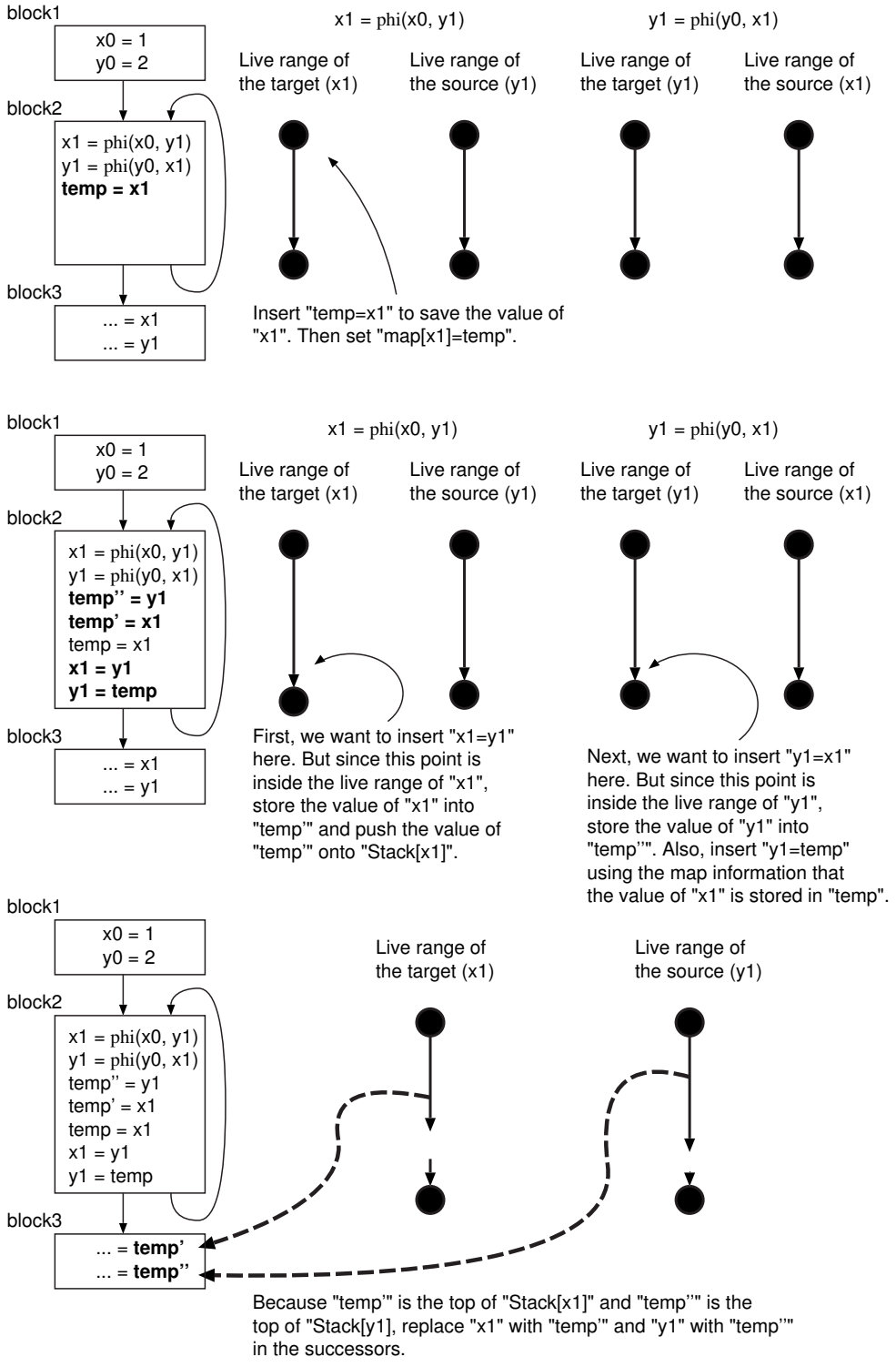


Figure 5.9: Translation process to solve the swap problem

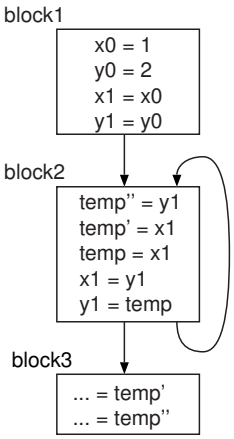


Figure 5.10: Result of translation for the swap problem

5.3.4 Algorithm

The algorithm of Briggs et al. is shown below with a brief explanation.

Algorithm (The algorithm of Briggs et al.)

```
/* The part which calls the translation from SSA form into normal form */
  analyze the live range
  for each variable v which appear in the program
    Stack[v] ← empty
  insert_copy( start )


/* procedure insert_copy( block ) */
  pushed ← null //the list of variables which are pushed on Stack
  for each instruction i in block
    if u is used, replace u by the top of Stack[u]
  schedule_copy( block )
  for each child of block in dominator tree
    insert_copy( child )
  for each variable n in pushed
    pop( Stack[n] )


/* procedure schedule_copy( block ) */
  /* Pass One: initialization of various data structures */
  copy_set ← ∅ //initialization of copy_set
  for each successor s of block
    j ← what number of argument in the φ-function in s the block is
    for each variable v appearing in the φ-function
      used_by_another ← false   //initialization of used_by_another
    for each φ-function in s
    //src: source, target: target
      src ← jth operand of φ-function
```

```
      copy_set ← copy_set ∪ {<src,target>}
      map[src] ← src
      map[target] ← target
      used_by_another[src] ← true

  /* Pass Two: put the pair for copy statements that do not require consider-
ation of the simultaneous assignment property to worklist */
  for each copy <src,target> in copy_set
    if used_by_another[target] ≠ true //if target is not the source of
other φ-functions
      worklist ← worklist ∪ {<src,target>}
      copy_set ← copy_set - {<src,target>}

  /* Pass Three: insert the copy statement considering the live range */
  while worklist ≠ ∅ or copy_set≠ ∅
    while worklist ≠ ∅
      Pick a <src,target> from worklist
      worklist ← worklist - {<src,target>}
      if the live range of target include the exit of block
        make a new variable temp,
        insert "temp ← target" to where the φ-function defining target was
        placed
        push temp on Stack[target]
        pushed ← pushed ∪ target
      insert "target ← map[src]" at the end of block
      map[src] ← target
      if src is in copy_set as target
        delete the pair from copy_set and put it into worklist, since
        the pair keeps the value of target
    if copy_set ≠ ∅
      Pick a <src,target> from copy_set
      copy_set ← copy_set - {<src,target>}
      make a new variable temp,
      insert "temp ← target" at the end of block
      map[target] ← temp
      worklist ← worklist ∪ {<src,target>}
```

5.3.5 Caution in implementation

The method of Briggs et al. implies that the copy statement might be inserted at the end of the block. However, if the implemented intermediate representation has conditional branch instruction or multi-way branch instruction, the copy statement will be inserted just before these instructions. In this case, there is a problem that if the target of the copy statement is used in these instructions, its value has been already updated before it is referred in the conditional expression, even though the live range of the target of the inserted copy statement does not include the exit of the block.

This problem is not pointed out in the documentation of the implementation of Briggs et al.[6].

To avoid such problem, we classify the situation into cases as below. Let v be the

target of the inserted copy statement, $LiveOut(B)$ be the set of variables which are live at the exit of block B to be inserted, and V be the set of variables which are used in the conditiononal expression at the end of B,

- if $v \in V$ and $v \notin LiveOut(B)$
  insert "$temp \leftarrow v$" into the point where the corresponding ϕ-function was defined, and replace v that is used in the conditional expression by $temp$
- if $v \in V$ and $v \in LiveOut(B)$
  replace v that is used in the conditional expression by $temp$, because "$temp \leftarrow v$" has already been inserted and $temp$ has been pushed on `Stack[`v`]`
- otherwise
  do nothing

Actually, we modified Pass Three of the algorithm (section 5.3.4) as follows.

Algorithm (Modification of algorithm of Briggs et al. (Pass Three))

```
/* Pass Three: insert the copy statement considering the live range */
while worklist ≠ ∅ or copy_set≠ ∅
  while worklist ≠ ∅
    Pick a <src,target> from worklist
    worklist ← worklist - {<src,target>}
    if the live range of target include the exit of block
      make a new variable temp,
      insert "temp ← target" to where the φ-function defining target was
      placed
      push temp on Stack[target]
      pushed ← pushed ∪ target

  === from here
      if target is used in the conditional expression in the block
        replace target in the conditional expression by temp
    else /* the live range of target does not include the exit of block */
      if target is used in the conditional expression in the block
        make a new variable temp, and insert "temp ← target"
        at the point where the φ-function defining target was placed
        replace target in the conditional expression by temp
  === to here

    insert "target ← map[src]" at the end of block
    map[src] ← target
    if src is in copy_set as target
      delete the pair from copy_set and put it into worklist, since
      the pair keep the value of target
  if copy_set ≠ ∅
    Pick a <src,target> from copy_set
    copy_set ← copy_set - {<src,target>}
    make a new variable temp,
    insert "temp ← target" at the end of block
```

```
map[target] ← temp
worklist ← worklist ∪ {<src,target>}
```

5.4 Method of Sreedhar et al.

The algorithm proposed by Sreedhar et al.[16] is a solution of the problem described in section 5.2. This section explains the algorithm of Sreedhar et al.

5.4.1 Principle

The principle of the back translation algorithm of Sreedhar et al. is quite different from the naive back translation algorithm or Briggs' back translation algorithm. The concept of Briggs' algorithm is that copy statements substitute for ϕ-functions. On the other hand the concept of Sreedhar's method is that ϕ-functions are removed by coalescing ϕ-functions.

First, Sreedhar's method focuses on the interference between the variables that appear in the ϕ-functions. In Figure 5.11 variables appearing in the ϕ-function are `a1, a2, a3` (including target). Here, we consider a set called **phiCongruenceClass**. The phiCongruenceClass is the set of variables appearing in the ϕ-function and having no interference among them. The initial state is set to

phiCongruenceClass[a1] $\leftarrow$ a1
phiCongruenceClass[a2] $\leftarrow$ a2
phiCongruenceClass[a3] $\leftarrow$ a3

We explain how to remove the interference between the live ranges later. In Figure 5.11, there is no interference among `a1, a2, a3`, so these variables can be in the same phiCongruenceClass.

phiCongruenceClass[a1] $\leftarrow$ {a1, a2, a3}
phiCongruenceClass[a2] $\leftarrow$ {a1, a2, a3}
phiCongruenceClass[a3] $\leftarrow$ {a1, a2, a3}

Since there is no interference among the variables in the phiCongruenceClass, these variables can be replaced by the same name. We can regard this as that the variables in the ϕ-function are coalesced. For example, these three PhiCongruenceClass which are described above are the same, so the left part of Figure 5.11 can be translated to the right part by replacing all these variables {a1, a2, a3} by A.

On the other hand, consider the case where ϕ-functions described below are included in the program,

a1 $\leftarrow$ ϕ(a2, a3) --- (1)
b1 $\leftarrow$ ϕ(a2, b3) --- (2)

Here, a2 belongs to both ϕ-function (1) and (2). We also coalesce the variables which belong to several ϕ-functions in the same way as above. That is, if variables `a1, a2, a3, b1, b3` do not interfere with each other, all these variables are coalesced.

Thus, basically new copy statements are not inserted in Sreedhar's method. However, if there are interferences among the variables in the ϕ-function, these variables cannot be coalesced. Therefore, copy statements are inserted to remove the interference between the variables which appear in the ϕ funtion (to make up phiCongruenceClass) in Sreedhar's method.

We explain how to reduce the interference in some detail below.

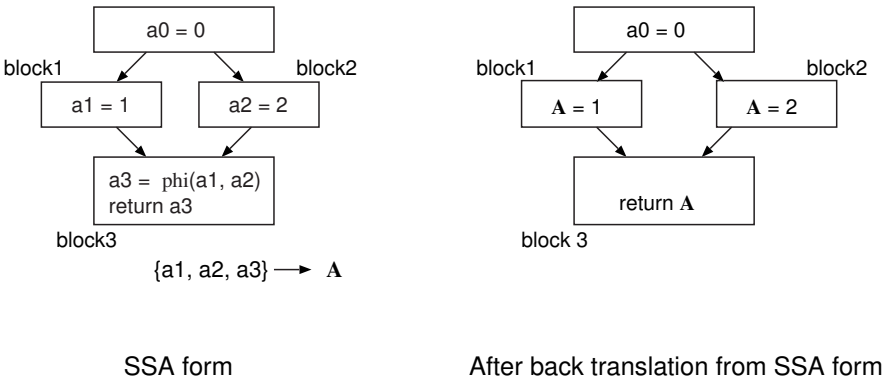


Figure 5.11: Principle of Sreedhar's method

5.4.2 Meaning of copy statements in Sreedhar's method

The meaning of the copy statements in Sreedhar's method can be understood by focusing on the property of ϕ-functions described below.

- The live range of the target of a ϕ-function always includes the entry of the block where the ϕ-function resides, but does not include the exit of the predecessor of the block.
- The live range of the source of a ϕ-function always includes the exit of the predecessor of the block containing the ϕ-function, but does not include the entry of the block where the ϕ function resides.

The role of inserting copy statement in Sreedhar's method is to rewrite the ϕ-function so as to shorten the live range of the target and sources in the ϕ-function as much as possible, when there are interference among the target and sources, or among sources in the ϕ-function. In practice, the live range is shortened as follows.

Copy statement for target

Assume we want to shorten the live range of the target of ϕ-function on the upper part of Figure 5.12 [2] . Then, replace the ϕ-function "`x2` $\leftarrow$ ϕ `(x1, x3)`" by "`x2'` $\leftarrow$ ϕ`(x1, x3)`", and insert "`x2` $\leftarrow$ `x2'`" just after the ϕ-function. Thus the live range of `x2'` which is the new target of the ϕ function becomes of minimum length as possible without changing the meaning of the program (the middle part of Figure 5.12).

Copy statement for source

Assume we want to shorten the live range of the second source of the ϕ-function at the upper part of Figure 5.12. Then, replace the ϕ-function "`x2` $\leftarrow$ ϕ`(x1, x3)`" by "`x2` $\leftarrow$ ϕ`(x1, x3')`", and insert "`x3'` $\leftarrow$ `x3`" to the end of the predecessor of the block. Thus the live range of `x3'` which is the second source of the new ϕ-function becomes of minimum length as possible without changing the meaning of the program (the lower part of Figure 5.12).

[2] The arrow describes the live range, • specifies that it includes the entry and exit of the block, ◦ specifies that it does not include the entry and exit of the block, if it is necsesary.

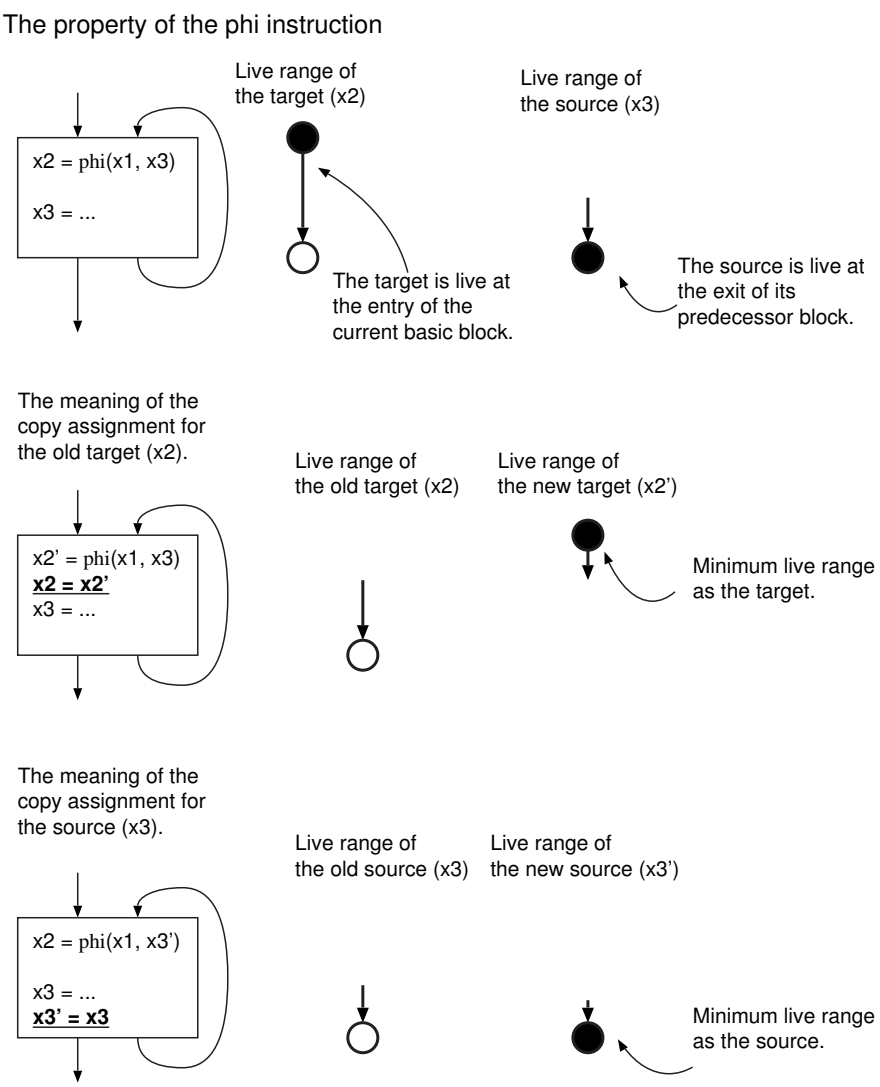


Figure 5.12: Meaning of copy statements in Sreedhar's method

Thus, the interference between the variables in the ϕ-function can be removed by appropriately inserting copy statements as described above. We give some examples in the next section.

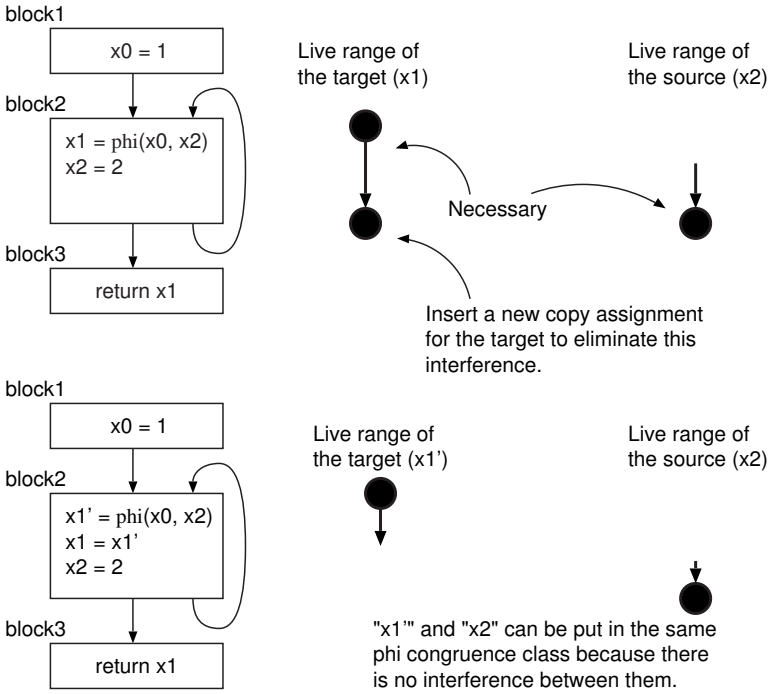


Figure 5.13: Translation process for the lost copy problem

5.4.3 Solution of the lost copy problem

Consider the upper graph in Figure 5.13 [3]. The variables which appear in the ϕ-function are {`x0, x1, x2`}. Considering the live ranges of `x1` and `x2`, `x1` and `x2` interfere with each other.

The way to remove such interference is to exclude the exit of `block2` from the live range of `x1` which is the target of the ϕ-function "`x1`$\leftarrow \phi$`(x0, x2)`". So we insert the copy statement for the target (the lower part of Figure 5.13).

Then, the variables {`x0, x1', x2`} which appear in the new ϕ funtion can be put in the same phiCongruenceClass, because they do not interfere with each other, The lost copy problem can be solved by replacing the variables in this phiCongruenceClass by X (Figure 5.14).

[3] We draw only the live range in the block `block2` which we should take notice with an arrow, and if it includes the entry or exit of the block, we specify it with $\bullet$. In the following part of this section, we use this convention. In addition, Figures describe the way of thinking, and do not correspond to the algorithm one by one.

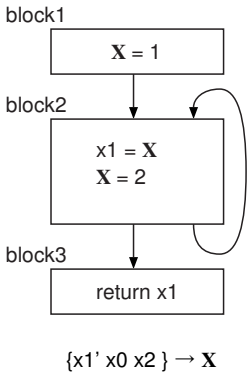


Figure 5.14: Result of translation for the lost copy problem

5.4.4 Solution of the simple ordering problem

Consider the upper part of Figure 5.15. The variables appearing in the ϕ-functions are the set {`x1, x0, x2`} and {`y1, y0, x1`}.

First, we consider the set {`x1, x0, x2`}. In the set {`x1, x0, x2`}, `x1` as the target interferes with `x2` as the source at the exit of `block2`. Since `x2` is a source, we cannot change the property that the live range of `x2` includes the exit of the block. So, to shorten the live range of the target, we insert the copy statement "`x1` $\leftarrow$ `x1'`" for the target at the beginning of `block2`, and replace the ϕ-function by "`x1'` $\leftarrow$ ϕ`(x0, x2)`" (the middle part of Figure 5.15). After that, there is no interference in the set of variables {`x1', x0, x2`} in the new ϕ-function "`x1'` $\leftarrow\phi$`(x0, x2)`".

Next, we consider the set {`y1, y0, x1`}. In the set {`y1, y0, x1`}, `y1` as the target interferes with `x1` as the source (the middle part of Figure 5.15). Since `x1` is a source, we cannot change the property that the live range of `x1` includes the exit of the block. So, to shorten the live range of the target, we insert the copy statement "`y1` $\leftarrow$ `y1'`" for the target at the beginning of `block2`, and replace the ϕ-function by "`y1'` $\leftarrow$ ϕ`(y0, x1)`". After that, there is no interference in the set of variables {`y1', y0, x1`} in the new ϕ-function "`y1'` $\leftarrow$ ϕ`(y0, x1)`" (the lower part of Figure 5.15).

Now, each of {`x1', x0, x2`} and {`y1', y0, x1`} can be collected as phiCongruence-Class. Then, after replacing {`x1', x0, x2`} by X and {`y1', y0, x1`} by Y, the ϕ-functions can be removed without changing the meaning of the program (Figure 5.16).

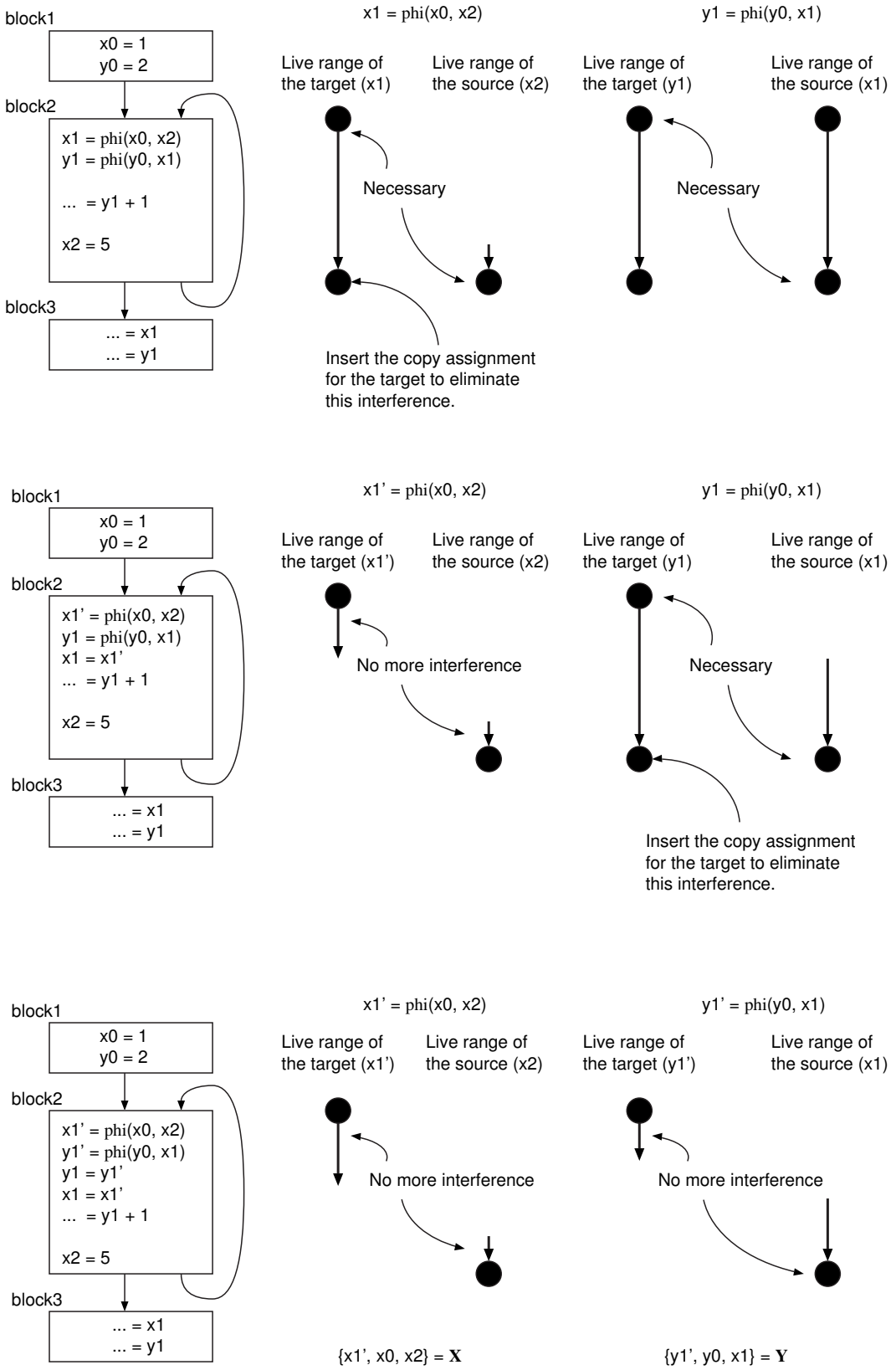


Figure 5.15: Translation process for the simple ordering problem

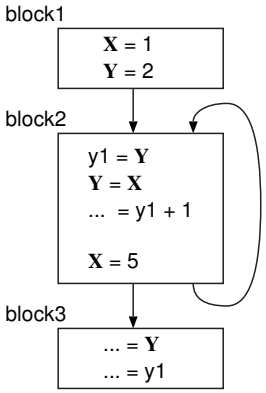


Figure 5.16: Result of translation for the simple ordering problem

5.4.5 The swap problem

Consider the upper part of Figure 5.17. The sets of variables appearing in the ϕ-functions are {x1, x0, y1} and {y1, y0, x1}.

First, we consider the set {x1, x0, y1}. In the set {x1, x0, y1}, x1 as the target interferes with y1 as the source at both the entry and exit of block2. This interference can be removed by insertion of the copy statement for x1 and the copy statement for y1. Since x1 is a target, we insert "'x1 ← x1'" at the beginning of block2, and replace the ϕ-function by "x1' ←ϕ(x0, y1)". On the other hand, since y1 is a source, we insert "y1' ← y1", and replace the ϕ-function by "x1' ←ϕ(x0, y1')". By these processes, there is no interference in the set variables {x1', x0, y1'} in the new ϕ-function (the middle part of Figure 5.17).

Next, we consider the set {y1, y0, x1}. y1 as the target interferes with x1 as the source in block2 (the middle part of Figure 5.17). In this case, the copy statement for y1 solves the problem. So, we insert the copy statement corresponding to the target y1. Insert "y1 ← y1''" at the beiginning of block2, and replace the ϕ-function by "y1'' ←ϕ(y0, x1)". After that, there is no interference in the set of variables {y1'', y0, x1} in the new ϕ-function. (the lower part of Figure 5.17).

By these processes, each of {x1', x0, y1'} and {y1'', y0, x1} can be collected as phiCongruenceClass. Then, after replacing {x1', x0, y1'} by X and {y1'', y0, x1} by Y, the ϕ-functions can be removed, solving the swap problem (Figure 5.18).

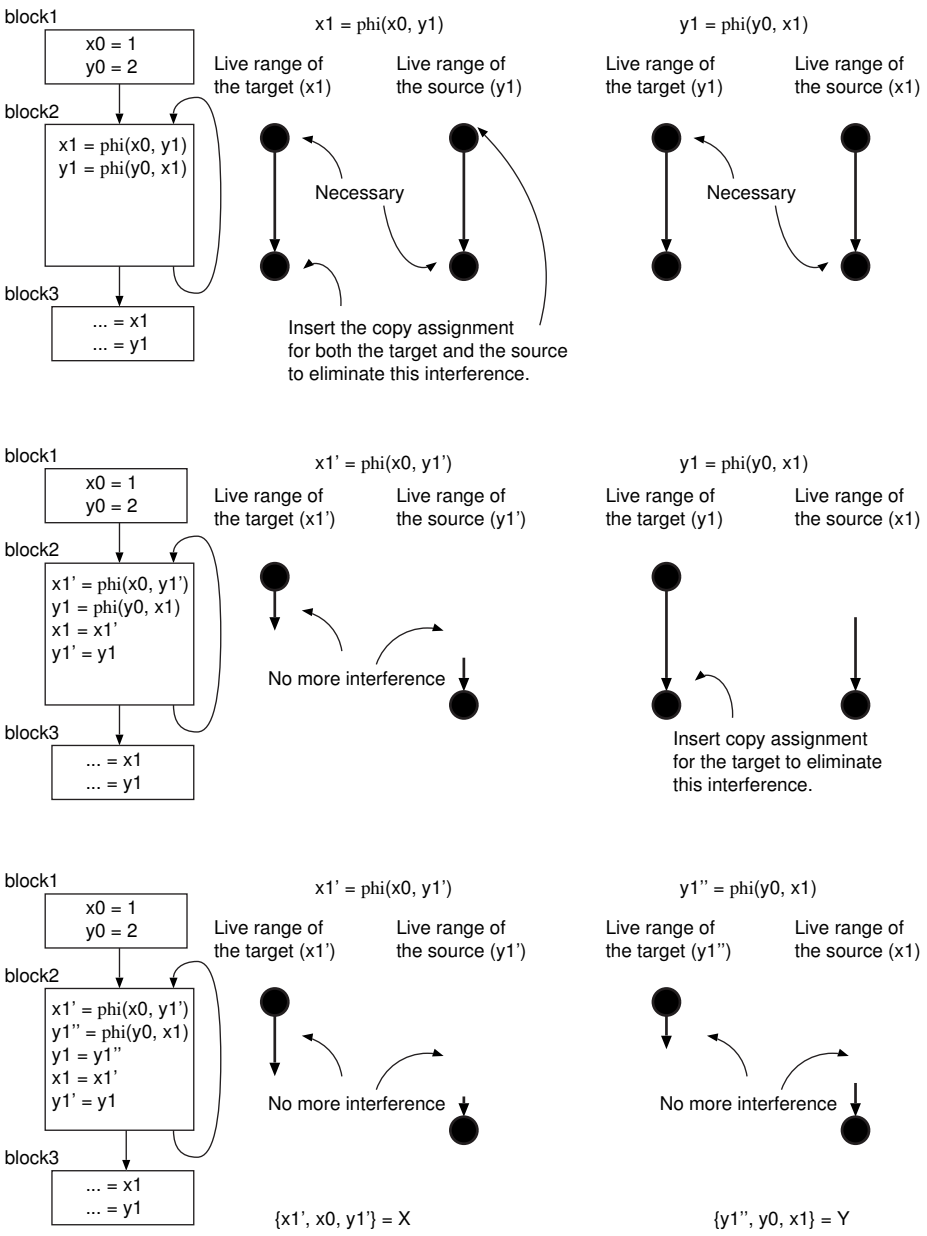


Figure 5.17: Translation process for the swap problem

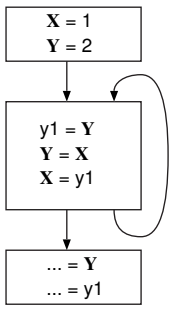


Figure 5.18: Result of the translation for the swap problem

5.4.6 Translation from TSSA Form into CSSA Form

The CSSA form (Conventional SSA form) is the SSA form directly created by the algorithm by Cytron et al. In the following the term *resources* mean variables. It is a characteristic feature of CSSA that the meaning of the original program is assured not to change when we replace all resources belonging to the same Phi Congruence Class with the same representative variable and we delete all ϕ-functions. However, an optimizing translation using the SSA form does not retain this feature of the CSSA form. That is, interference occurs between resources of ϕ-function. Here, interference refers to the situation when the live ranges of two variables a and b overlap [1, 3, 7]. This type of SSA form is referred to as a TSSA form (Transformed SSA form).

Figure 5.19 shows an example of a CSSA form. Variables x1, x2 and x3 belonging to the same Phi Congruence Class can be replaced by the representative variable x. Figure 5.20 shows the normal form resulting when variables x1, x2 and x3 in Figure 5.19 are replaced by the representative variable x and the ϕ-function is eliminated.

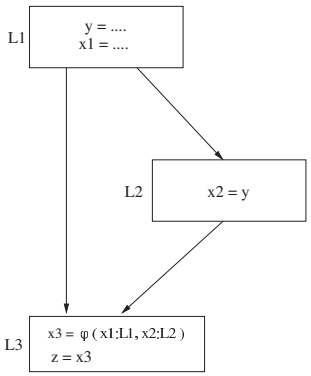


Figure 5.19: Example of SSA form

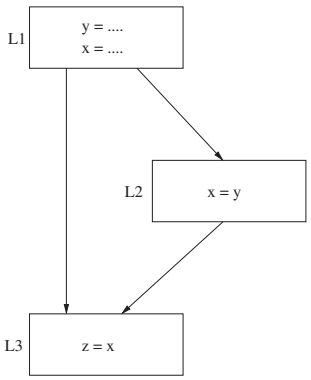


Figure 5.20: Translation example from CSSA to normal form

On the other hand, if we coalesce the assignment statement x2←y in Figure 5.19, we get Figure 5.21.

The variables x1, x3 and y in Figure 5.21 belong to the same Phi Congruence Class. However, if these variables are replaced with the representative variable x, the meaning of the original program changes. This is because deletion and coalescing of the copy statement x2←y has caused the live ranges of x1 and y to interfere. This type of TSSA

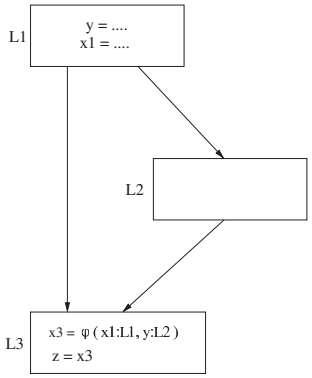


Figure 5.21: TSSA form example

form can be translated into CSSA form by inserting copy statements.

The steps for translating from the TSSA form into normal form are as follows.

1. Translate TSSA form into CSSA form
2. Delete unnecessary copy statements
3. Delete ϕ-functions and translate into normal form

In the following, the description of the method for implementing step 1 is given. The method for performing step 2 is described in 5.5. Note the following in implementing step 3. When resources of ϕ-functions are replaced with representative variables, if the same Phi Congruence Class includes resources of multiple ϕ-functions, those resources must be replaced by the same representative variable at once. All other processing are trivial and will be omitted.

First, we describe the preprocessing for 1.

Preprocessing for ϕ-function

The Algorithm described at [16] by Sreedhar et al. considers only the case where source resources of ϕ-functions are variables. However, if this system optimize the program, the source resources of ϕ-functions may become constants during the process of optimization. As described above, in the method of Sreedhar et al., we set up the Phi Congruence Class from the live range of the variables and translate to the normal form. However if the resource of a ϕ-function is a constant, there are some problems that how the live range or Phi Congruence Class of the constant should be treated and so on. Therefore, our system checks the source resources of ϕ-functions just before translation to normal form using the method of Sreedhar et al. If any source resource of a ϕ-function is a constant, we make a new variable `temp`, and insert copy statement "`temp` $\leftarrow$ *constant*" into the predecessor. After that, the constant of the source resource of ϕ-function is replaced by variable `temp`.

Preprocessing for temporary variables

When optimization is performed using our system, for example, if you break down the expression to three-address code using the method described in section 6.9, a lot of temporary variables are generated. The meaning of the program is not changed by using

temporary variables, but they might hinder to generate efficient code in the code generator. Therefore, our system removes the temporary variable if it satisfies the condition described below just before translating SSA form to normal form.

- The live range of the variable is in the basic block (basic block local), and it is referred only once.

The removal of such temporary variables is implemented by substituting the part where the temporary variable is used with the right hand side expression which defines the temporary variable.

(Example) Suppose *temp* is basic block local.

$$
\begin{array}{lll}
temp & \leftarrow & a+b \\
 & & ... \qquad (temp \text{ is not used here}) \\
c & \leftarrow & temp + ... \\
 & & ... \qquad (temp \text{ is not used here})
\end{array}
$$

In the above case, "$c \leftarrow temp + ...$" is replaced by "$c \leftarrow (a+b) + ...$".

Method I

In Method I, copy statements are inserted for all ϕ-function resources. There are two types of function resources, resources that are defined by the ϕ-function, which are called destination (target) resources, and resources that are ϕ-function parameters, which are called source resources. We insert a copy statement for each destination resource into the same basic block as the ϕ-function, and insert a copy statement for each source resource in the predecessor block for that resource. When a copy statement is to be inserted into the same basic block as the ϕ-function, we place it directly after the ϕ-function (after all ϕ-functions if there are multiple ϕ-functions). When a copy statement is to be inserted in the predecessor block, it should be inserted after the last instruction of the predecessor (immediately before the last instruction, if that instruction is a conditional branch instruction) [4]. Figure 5.22 shows an example of Method I.

Figure 5.22(a) shows an example of a TSSA form and (b) shows use of Method I to translate (a) into CSSA form.

Method II

In Method II, copy statements are inserted only when ϕ-function resources interfere with each other. In Figure 5.22 (a), resources x1 and x2 interfere with each other. This requires that copy statements for x1 and x2 should be inserted to eliminate interference. x3 does not interfere with x1 or x2 so no copy statement is required.

Insertion of a copy statement requires that the interference graph and Phi Congruence Class should be updated.

Method III

Method III uses the liveness information of variables in the program. We present Method III using Figure 5.22.

[4] Although Sreedhar et al. specify the insertion of copy statements in basic blocks, they do not specify placement within a basic block. This may cause problems if there are any critical edges. For details, see Section 5.4.6.

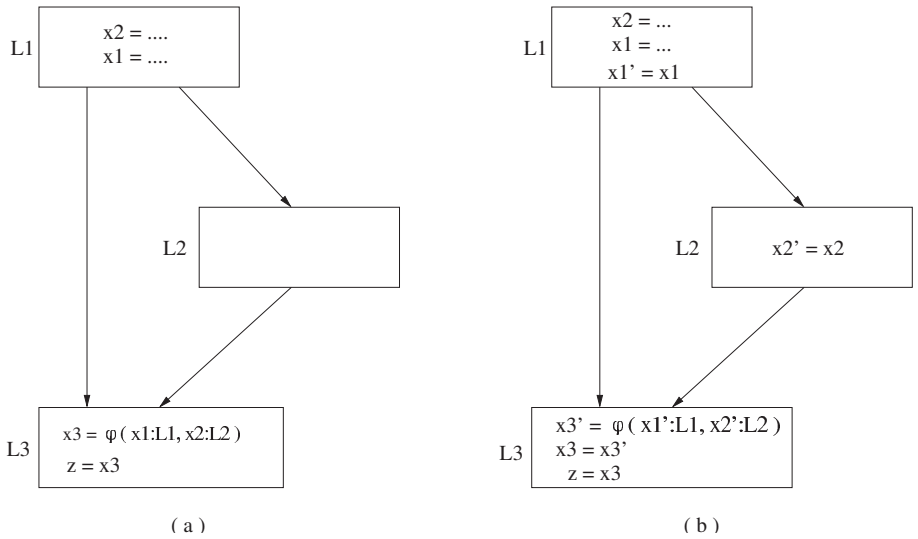


Figure 5.22: Example of translation from TSSA form (a) to CSSA form (b) using Method I

In Figure 5.22 (a), ϕ-function resources x1 and x2 interfere with each other. The liveness information indicates that LiveOut [L1] is {x1,x2}, and that LiveOut[L2] is {x2}. Since x2 is included in LiveOut[L1], insertion of a copy statement x1'←x1 at the end of block L1 only does not eliminate interference between ϕ-function resources (in this case, insertion of a copy statement in L2 is also required). On the other hand, x1 is not included in LiveOut[L2], so the insertion of copy statement x2'←x2 in L2 is enough to eliminate interference between ϕ-function resources. In Method III, the places for copy statement insertion to eliminate interference between ϕ-function resources are selected to minimize the use of copy statements. Thus, a copy statement is inserted only for x2 in the above example.

Whereas LiveOut is used to eliminate interference between ϕ-function resources, LiveIn and LiveOut information are required to eliminate interference between destination and source resources. The copy statements inserted by Method III are known that they can not be removed using standard coalescing algorithms based on the interference graph. However, there is no guarantee that only the minimum number of copy statements are inserted.

The following is the description of the algorithm of Method III . Here, $y \in pcc(x)$ describes that variable y belongs to Phi Congruence Class of x.

1. for each x (x is the ϕ-function resource in the current CFG) do
   pcc(x)←x
   end for
   /*
   This algorithm processes ϕ-functions one by one.
   Steps 2 to 4 described below are executed for each ϕ-function.
   */

2. for each ϕ-function in the current CFG do
   /*
   It is assumed that ϕ-functions are in the following format.

x0←f(x1:L1, x2:L2, ..., xn:Ln)
That is, we assume in the following that a ϕ-function includes the left-hand side target.
L0 is the basic block containing the ϕ-function.
*/

3. Set candidateResourceSet{} for the corresponding ϕ-function
for each xi (xi is a ϕ-function resource, $0 \leq i \leq n$) do
unresolvedNeighborMap(xi)←{}
end for
/*
candidateResourceSet{} holds variables for inserting copy statements.
unresolvedNeighborMap maintains variables for deferring insertion when we can not uniquely decide which resources need insertion of copy statements.
*/

4. for each xi:Li and xj:Lj
(xi, xj are ϕ-function resources currently handled, $0 \leq i, j \leq n \wedge xi \neq xj$) do
Check interference information
if ($yi \in pcc(xi) \wedge yj \in pcc(xj) \wedge$ yi and yj interfere with each other) then
to break the interference between xi and xj, determine where to insert copy statements according to the four cases given below
/*
Do not check the interference for combinations of pcc(xi) and pcc(xj) if the two are the same.
(This is not described in the original paper.)
These is a feature of the Phi Congruence Class that variables that belong to the same Phi Congruence Class
can be replaced by the same variable name.
Thus in this case it is determined that xi and xj will have the same variable name afterwards.
So, if we test interference under these conditions they will inevitably indicate interference,
and results in the insertion of unnecessary copy statements.
Use LiveOut to check interference for source resources
and use LiveIn for destination resources.
When two resources interfere,
the decision of copy statement to be inserted is
determined according to the following four cases.
(The description indicated as Live may be either LiveIn or LiveOut)
Case 1:
The intersection of pcc(xi) and Live(Lj) is not empty,
and the intersection of pcc(xj) and Live(Li) is empty.
xi is added to candidateResourceSet{}, to insert copy statement xi'←xi into Li.
Case 2:
The intersection of pcc(xi) and Live(Lj) is empty,
and the intersection of pcc(xj) and Live(Li) is not empty.

```
            xj is added to candidateResourceSet{}, to insert copy statement
            xj'←xj into Lj.
          Case 3:
            The intersection of pcc(xi) and Live(Lj) is not empty.
            and the intersection of pcc(xj) and Live(Li) is not empty.
            xi and xj are added to candidateResourceSet{},
            to insert copy statement xi'←xi into Li
            and copy statement xj'←xj into Lj.
          Case 4:
            The intersection of pcc(xi) and Live(Lj) is empty,
            and the intersection of pcc(xj) and Live(Li) is empty.
            Insertion of a copy statement into either block
            can eliminate the interference. Thus the decision is
            deferred, and xj is added to unresolvedNeighborMap(xi),
            and xi is added to unresolvedNeighborMap(xj).
        */
      end if
    end for
```

5.

```
    /*
      Process the unresolved resources generated by Case 4
    */
    for each v (v is a variable with an unresolvedNeighborMap) do
      if v ∉ candidateResourceSet{} then
        for each xi ∈ unresolvedNeighborMap(v) do
          if xi ∉ candidateResourceSet{} then
            add v to candidateResourceSet{}
          end if
        end for
      end if
    end for
```

6.

```
    /*
      Insert copy statement
    */
    for each xi ∈ candidateResourceSet{} do
      if xi is a source resource of a φ-function then
        for each Lk (Lk is the predecessor coresponding to xi) do
          insert a copy: xnew_i←xi at the end of Lk
          /*
            The original paper just states that insertion occurs at the end of Lk, but
            actually if there is a conditional branch instruction, the copy statement
            should be inserted before that.
          */
          Replace xi used in the current φ-function source by xnew_i
          Create pcc(xnew_i) and insert xnew_i there
          Add xnew_i to LiveOut(Lk)
          /*
            If Lj ∈ succ(Lk) ∧ xi ∉ LiveIn(Lj) ∧
```

```
        xi is not used in φ-function associated with Lk in Lj,
        delete xi from LiveOut(Lk).
      */
      remove←true
      for each Lj ∈ succ(Lk) do
        if xi ∈ LiveIn(Lj) then
          remove←false
        end if
        if xi is in the source resource of φ-function in Lj,
          and the predecessor corresponding to xi is Lk then
          remove←false
        end if
      end do
      if remove is true then
        remove xi from LiveOut(Lk)
      end if
    end for
  else /* xi is the destination (target) resource of a φ-function */
    insert a copy: xi←xnew_i at the beginning of L0
    /*
      The original paper just states that insertion occurs at the beginning of L0,
      but actually the copy statement should be inserted at the end of all
      φ-functions
    */
    Replace xi used in the target of the current φ-function with xnew_i
    Create pcc(xnew_i) (and add xnew_i there)
    Remove xi from LiveIn(L0)
    Add xnew_i to LiveOut(L0)
  end if
  Update the interference graph
end for
```

7.

```
/*
  Merge Phi Congruence Class of all resources of φ-function
*/
currentPhiCongruenceClass←{}
for each xi (xi is a resource of a φ-function, 0 ≤ i ≤ n) do
  Add pcc(xi) to currentPhiCongruenceClass
  /*
    If pcc(yj) is not added to currentPhiCongruenceClass, where yj is ∈ pcc(xi),
    then add it too (Note 1).
    This is not described in the algorithm of the original paper.
  */
  for each yj ∈ pcc(xi) do
    if pcc(yj) is not added to currentPhiCongruenceClass then
      add pcc (yj) to currentPhiCongruenceClass
    end if
  end for
end for
```

```
    for each xi ∈ currentPhiCongruenceClass do
      pcc(xi)←currentPhiCongruenceClass
      /*
        This is not a copy, but make the left-hand side points (pointer reference) to
        the right-hand side (Note 2).
      */
    end for
  end for /* end of "for each φ-function" of Step 2 */
```

8. for each x (x is a resource of a ϕ-function) do
   if pcc(x) includes only one resource then
     pcc(x)←{}
   end if
   end for

(Note 1) Consider a program with multiple ϕ-functions. First, process

$$x1 \leftarrow \phi(x2, x3)$$

Then after processing Step 7

$$\begin{aligned} pcc(x1) &= pcc(x2) \\ &= pcc(x3) \\ &= \{x1, x2, x3\} \end{aligned} \tag{5.1}$$

hold. Then process

$$x4 = \phi(x5, x1)$$

If (Note 1) section of the algorithm is omitted, after completing Step 7,

$$\begin{aligned} pcc(x4) &= pcc(x5) \\ &= pcc(x1) \\ &= \{x4, x5, x1\} \end{aligned} \tag{5.2}$$

hold. However, this is incorrect. The correct result should be

$$\begin{aligned} pcc(x1) &= pcc(x2) \\ &= pcc(x3) \\ &= pcc(x4) \\ &= pcc(x5) \\ &= \{x1, x2, x3, x4, x5\} \end{aligned} \tag{5.3}$$

This is performed by the (Note 1) section. In general, the property holds that a resource does not belong to multiple Phi Congruence Classes. This is because a Phi Congruence Class corresponds to the transitive closure of a union of sets.
For example, resource x1 does not belong to Phi Congruence Class of both (5.1) and (5.2).

(Note 2) In this algorithm, each ϕ-function is processed one by one. Consider the same example as described in (Note 1). To ensure that the Phi Congruence Class becomes as described in (5.3), all of pcc(x1), pcc(x2), pcc(x3), pcc(x4), pcc(x5) including previously processed ϕ-function resources x2, x3 are made to point to the same {x1,x2,x3,x4,x5} object (list) in this implementation. Thus the same Phi Congruence Class has a single entity and a resource does not belong to multiple Phi Congruence Class entities.

Problems in SSA Backtranslation on LIR

The following is common to all translation from SSA form to normal form and is not limited to the method used by Sreedhar et al. If the basic block containing ϕ-function is B, a copy statement for the source resource of the ϕ-function is inserted in P $\in$ pred(B) and the ϕ-function is eliminated. These copy statements are often placed at the end of instruction sequence in P. However, LIR places JUMP/JUMPC/JUMPN instructions to jump to a successor at the end of the basic block. Then the copy statement should be naturally inserted before the JUMP/JUMPC/JUMPN instruction, but this is also the instruction where variables are referenced. Thus the insertion of a copy statement may change the value of the variable that this instruction references, thereby may corrupt the meaning of the program.

Let the basic block containing JUMPC/JUMPN be Bj. The existence of a ϕ-function in Bsj $\in$ succ(Bj), means in general that Bj $\longrightarrow$ Bsj is a critical edge. A conservative approach would be to remove the critical edges prior to translation from SSA form to normal form. However, all critical edges may not cause the above phenomenon. For example, the method used by Sreedhar et. al. can often safely handle critical edges. However, we have confirmed that they cannot be handled safely under specific conditions.

For more information concerning critical edges, see section 6.12.

5.5 Coalescing

When normal form LIR is translated to SSA form LIR, a single variable generally becomes multiple SSA variables. [5] A back translation of the SSA form generates multiple copy statements. Coalescing is the most effective method to assign one variable to the variables associated with these copy statements and remove the copy statements. Coalescing removes the copy statements and handles the source variable of a copy and the target variable of the copy as one and single variable. Of the many coalescing methods that have been proposed so far, we have implemented two in this system : Chaitin's method and the method by Sreedhar et al. Either of these can be applied by specifying options.

In this system, Chaitin's method is applied to normal form LIR after SSA back translation while the Sreedhar's method is applied at the time of SSA back translation.

5.5.1 SSA-based Coalescing

An example of an SSA form program is given in Figure 5.23. This SSA form program is in CSSA form, so ϕ-functions can be removed. Figure 5.24 shows the normal form program when the ϕ-functions have been removed.

The live ranges of X and Y in copy statements X$\leftarrow$Y in the program shown in Figure 5.24 overlap, so Chaitin's coalescing method cannot be used. Sreedhar et al. proposed a coalescing method in a CSSA form program that can remove copy statement X1$\leftarrow$Y1

[5] An SSA variable is a variable that enables static single assignment.

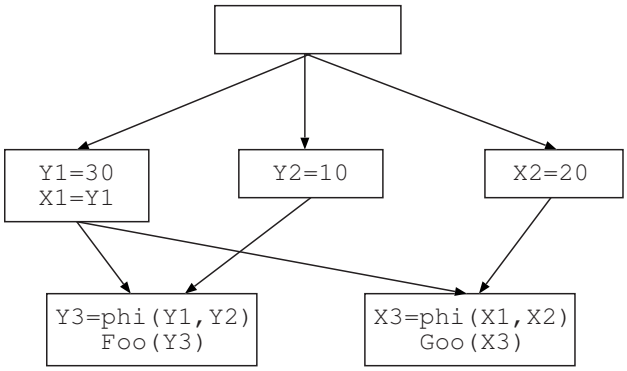


Figure 5.23: Exampl of SSA form (CSSA form) program

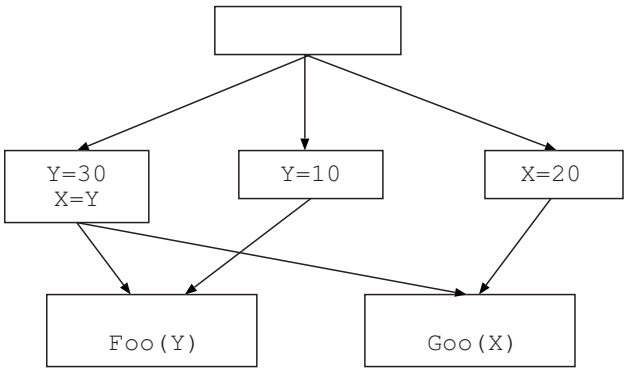


Figure 5.24: Figure 5.23 Translating SSA form (CSSA form) programs to normal form

shown in Figure 5.23 [16]. This method is used within the SSA back translation proposed by Sreedhar et al. and can be used to remove copy statement X←Y under certain conditions even if the live ranges of two variables interfere.

The method proposed by Sreedhar et al. is described below. Consider the copy statement X←Y, where X and Y do not belong to the same Phi Congruence Class. Consider the following three cases when coalescing the copy statement X←Y.

Case 1: When pcc(X)={ } ∧ pcc(Y)={ }
This means that X and Y is not referenced by any ϕ-function. This copy statement can be removed even if their live ranges interfere.

Case 2: When pcc(X)={ } ∧ pcc[Y] ≠ { }
If the live range of X interferes with any resource in the set {pcc(Y) − Y} then X←Y cannot be removed. Otherwise, it can be removed.
The case where pcc(X) ≠ { } ∧ pcc(Y)={ } is the reverse of this case.

Case 3: When pcc(X) ≠ { } ∧ pcc(Y) ≠ { }
If the live range of a resource in pcc(X) interferes with that of a resource in the set {pcc(Y) − Y}, or if the live range of a resource in pcc(Y) interferes with that of a resource in the set {pcc(X) − X}, X←Y cannot be removed. Otherwise, it can be removed.

Consider the example in Figure 5.23. From the figure, we see that pcc(X1)={ X1,X2,X3

} and that pcc(Y1)={ Y1,Y2,Y3 }. The live range of X1 and Y1 in the copy statement X1←Y1 interferes, but Case 3 can be applied to remove X1←Y1. After the copy statement X1←Y1 is removed, pcc(X1) and pcc(Y1) are merged.

5.5.2 Chaitin's Method

When the live ranges of two variables a and b overlap, or more accurately, when that is an assignment of a variable to the other when one variable is live, a and b are said to interfere [1, 3, 7]. The coalescing method proposed by Chaitin [7] is to always remove copy statement a←b when a and b in the copy statement a←b do not interfere with each other. Chaitin's method is therefore referred to as being an aggressive method.

The method proposed by Chaitin in [7] was originally a method for register allocation. Therefore the number of registers in this system are set to ∞ and coalescing is repeated while it can be done. Figure 5.25 shows Chaitin's coalescing algorithm.

```
/* Step 1 */
for each block X do
   for each LIR node do
      if node is a copy statement and the source and the
      target of the copy do not interfere with each other then
         Add the copy statement to the table
         Remove the added copy statement from LIR
      end if
   end for
end for

/* Step 2 */
for each block X do
   for each LIR node do
      if the node uses the left-hand side value of a copy statement then
         Replace the left-hand side value of the copy statement used in
         the node by the right-hand side value of the copy statement
         (If the copy is in chain [a], the process continues until the root
         is reached)
      end if
   end for
end for
```

[a] In "chain" means replacement of z in x←y; z←x; not by x, but by y.

Figure 5.25: Chaitin's coalescing algorithm

Chaitin's coalescing can be executed after any SSA back translation method is applied. However, as described in section 5.4.6, it is known that the copy statements inserted in Sreedhar's back translation Method III cannot be removed by coalescing based on interference graph. Consequently, Chaitin's coalescing seems to have no effect after the execution of the back translation of Sreedhar's Method III.

Chapter 6

Optimizations on SSA Form

This chapter describes optimizations on SSA form. Optimizations on SSA form implements the following.

- Copy propagation
- Common subexpression elimination
- Global value numbering and partial redundancy elimination based on question propagation
- Moving loop invariant expressions out of loops
- Operator strength reduction and test replacement for induction variables
- Constant propagation considering conditional branch
- Dead code elimination
- Algorithms for constructing SSA graph and for finding equality of variables
- Dividing into three-address code
- Elimination of empty blocks
- Concatenate basic blocks
- Elimination of critical edges
- Global reassociation
- Removing useless ϕ instructions

This system uses an independent pass for each optimization. Depending on COINS compiler driver option, it is possible to perform any optimization in any order and any number of times.

Each optimization is described in the following.

6.1 Copy Propagation

Copy propagation has the same effect as copy folding described in section 4.2.2. Thus it is a method for eliminating copy statements like a←b in the source code. Copy propagation checks the live range of each variable and replaces uses of the target variable (a of a←b) following the copy statement by the source variable (b of a←b), if the target variable is live (i.e. there are no new assignment statements for a). However, variable values in SSA form are assured to remain statically single, thus there is no need to check the live ranges of variables and they can be simply replaced. Whereas the process described in section 4.2.2 is performed during SSA translation, copy propagation described here is an independent process that is used as an optimization.

Copy propagation involves two steps. In Step 1, copy statements are detected and in Step 2 the variable names are changed. The copy propagation algorithm is shown in Figure 6.1.

```
/* Step 1 */
for each block X do
  for each LIR node do
    if node is a copy statement then
      Add copy statement to table
      Delete the added copy statement from LIR
    end if
  end for
end for

/* Step 2 */
for each block X do
  for each LIR node do
    if the node uses the left-hand side value of a copy statement
     then
      Replace the left-hand side value of the copy statement used
      in the node by the right-hand side value of the copy
      statement
      (If the copy is in chain [a] the process continues until the
       root is reached)
    end if
  end for
end for
```

[a] In "chain" means replacement of z in x←y; z←x; not by x, but by y.

Figure 6.1: Copy Propagation Algorithm

6.2 Common Subexpression Elimination

Consider that an expression X op Y (expression of operation on X, Y) is calculated at a point in a program. If that expression is known to be always calculated before reaching that point, then the calculation of that expression can be omitted. To use data flow equations to obtain redundant expressions, it is necessary to obtain the expressions that are calculated or used at a given point. We describe an algorithm for eliminating common subexpressions generated by the SSA form in the following [13].

Consider the following program.

```
1) a = b + c
2) d = b
3) e = b + c
4) a = c + d
```

Translate this program to SSA form.

```
1)  a0 = b0 + c0
2)  d0 = b0
3)  e0 = b0 + c0
4)  a1 = c0 + d0
```

We prepare a table showing the relationship between expressions and variable names and a table showing the relationship between variables and their copies. Processing statements 1) and 2) will result in a table like Table 6.1.

Table 6.1: Common subexpression elimination : after processing 1) and 2)

expression	variable name
b0+c0	a0

variable name	variable name
d0	b0

We know from Table 6.1 that in processing 3) the right-hand side expression is already computed and has the value a0 and that in processing 4) similar fact holds since d0 is a copy of b0. Thus after processing 3) and 4), the table showing the relation between expressions and variable names and the table showing the relation between variables and their copies will be as shown in Table 6.2.

Table 6.2: Common subexpression elimination: after processing 3) and 4)

expression	variable name
b0+c0	a0

variable name	variable name
d0	b0
e0	a0
a1	a0

The program after common subexpression elimination is as follows.

```
1)  a0 = b0 + c0
2)  d0 = b0
3)  e0 = a0
4)  a1 = a0
```

Eliminating common subexpressions in SSA form can be done for the entire program by using ϕ-functions at merge points. The value of a variable is transferred to a block either from a block that directly dominates that block or from a ϕ-function at the head of the block. Thus the processing works better if we start from the root of the dominator tree and proceed to the leaves.

An algorithm for eliminating common subexpressions is shown in Figure 6.2. The relation between an expression and a variable name is indicated as "variable name ←NameExp (expression)". The relations between variable names when a1 is a copy of a0 are indicated as "a1←Copy(a0)". "V←Copy(V1)" in Figure 6.2 means that the fact that V is a copy of V1 is written to the table. Similarly, "V←NameExp(e)" means that expression e and its corresponding variable name V is written to the table. The ϕ-function is also considered to be an expression.

In the current implementation, given an expression V←X op Y, we write the following in the table: V on the left-hand side that is a REG expression and X op Y on the right-hand side that does not include MEM expressions. If the right-hand side X op Y is a constant or a function call, it is not written to the table. However, if the simple alias analysis described in Chapter 7 is performed, we also write the right-hand side X op Y that includes MEM expressions to the table.

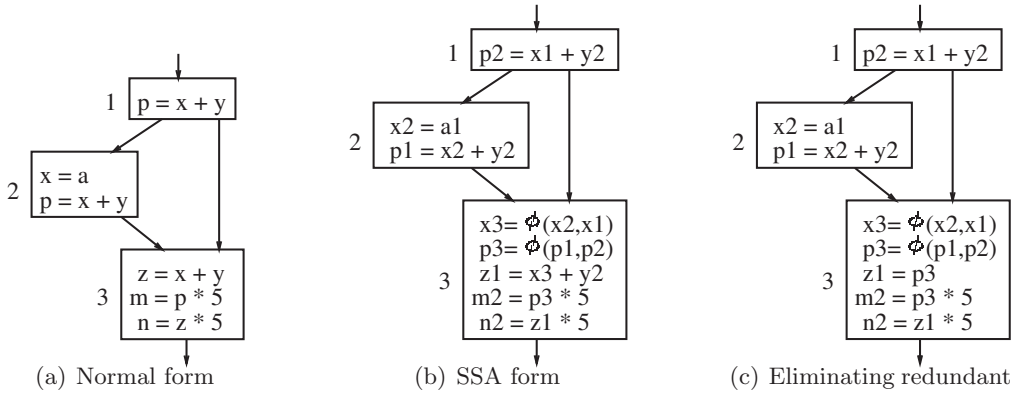
```
cse(block B) /* procedure cse */
  for each φ-function in B : V← φ(···) do
    if φ-function e which is the same as the right-hand side is in table then
      eliminate this statement and do V←Copy(NameExp(e))
    else if φ-function parameters are the same variable V1 then
      eliminate this φ-function and do V←Copy(V1)
    else if φ-function parameters are the same constant C then
      eliminate this φ-function
      and insert V←C immediately after all φ-functions
      which are at the beginning of the block
    end if
  end for
  for each statement S in B do
    if S is a define statement not containing a CALL expression: V←X op Y then
      if S contains a variable X s.t. X←Copy(X0) then
        replace X by X0
      end if
      if S contains subexpression X op Y where Z←NameExp(X op Y) exists then
        replace subexpression X op Y by Z
      end if
      if S is copy statement V←V1 then
        eliminate S and do V←Copy(V1)
      end if
      if V is a REG expression then
        if no MEM expressions is in the right-hand side e then
          do V←NameExp(e)
        else if alias analysis is performed then
          do V←NameExp(e)
        end if
      end if
    else /* if S is define statement including a CALL statement or a statement other
      than a define statement */
      if S contains use of variable X s.t. X←Copy(X0) then
        replace X by X0
      end if
      if S contains a subexpression X op Y where Z←NameExp(X op Y) exists then
        replace subexpression X op Y by Z
      end if
    end if
  end for
  for each block C ∈succ(B) do
    for each φ-function in C do
      if V exists in φ-function parameters s.t. V←Copy(V1)
      replace it by V1
    end for
  end for
  for each block C ∈ domChild(B) do
    cse(C)
  end for
  Remove items written to correspondence tables in block B
```

Figure 6.2: Algorithm for common subexpression elimination

6.3 Global value numbering based on question propagation

Some popular code optimizations like removal of common subexpression require computing *availability* of a value generated by expression e located at program point p, which shows that there are other expressions to evaluate the same value as e on every path from the program entry to p.

Traditionally, such information is given by finding *available expressions* [1]. We say that expression e is *available* at program point p, if there are lexically same occurrences as e on every path from the program entry to p, and there is no assignment to e's operands on path from the last occurrence of e to p. It is known well that the available expressions can be efficiently computed by dataflow analysis using bit-vector representation.



(a) Normal form (b) SSA form (c) Eliminating redundant expressions

Figure 6.3: Availability in SSA form

On the other hand, for programs in *static single assignment* (SSA) form [11, 15], we can find available expressions more easily, because lexically same occurrences of an expression are guaranteed to have the same value without taking account of assignments to their operands. However, some expressions may not be found to be available even if they are available in normal form, because originally same expressions may have lexically different representations in SSA form. For example, in Figure 6.3(a), `x + y` is available at entry of node 3. However, in Figure 6.3(b) `x3 + y2` corresponding to `x + y` at node 3 cannot be found to be available, because `x1 + y2` at node 1 and `x2 + y2` at node 2, which are available as `x + y` at entry of node 3 in Figure 6.3(a), are lexically different from `x3 + y2` whose operand `x3` is defined by ϕ-function.

For such available expression problem in SSA form, Rosen *et al.* proposed *question propagation* method [14], which collects answers to question "Is an expression e available here?", backwardly asking the questions at each node on paths to a specified point. When the question reaches ϕ-function defining e's operands during propagation, they are renamed to each ϕ-function's argument corresponding to predecessors to be propagated. For example, in Figure 6.3(b), to propagate question about `x3 + y2` at node 3, it is transformed into two questions by renaming its operand `x3` to `x2` and `x1` respectively, and they are propagated to proper nodes. After that, it is found that `x2 + y2` is available at node 2 and `x1 + y2` is also available at node 1, because there is an occurrence of the inquired expression at each node. Consequently values generated by `x3 + y2` are found to be available at node 3 so that it can be eliminated by replacing it with `p3` as shown by Figure 6.3(c). However, applying such question propagation to all occurrences of expressions is not as efficient as dataflow analysis approach, so most previous works have devoted to the

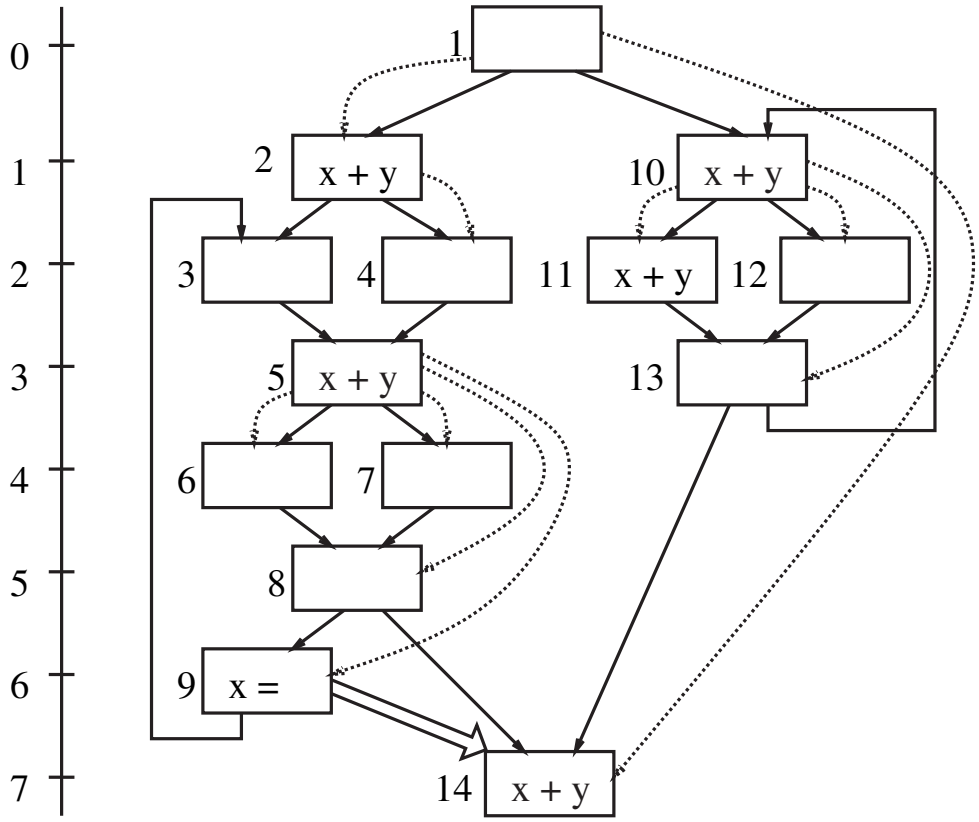


Figure 6.4: Sparse edges and virtual edges

lexically same expressions as available expressions as well as dataflow approach.

We have improved the question propagation in the following points:

1. suppressing unnecessary propagations from node v to its predecessors on *control flow graph* (CFG), directly propagating the questions to v's dominator d, and

2. detecting partially redundant expressions and eliminating them.

For the ease of presentation, we use a special graph representation, called *propagation graph*, which is composed of CFG nodes, CFG edges and sparse edges whose sources dominate their destinations. The sparse edge means propagating questions from its destination to sources holds program semantics, where note that it does not guarantee any effectiveness. Figure 6.4 illustrates the propagation graph, in which sparse edges are shown by dotted arrows, while CFG edges are shown by solid arrows.

In the following, an outline of global value numbering and partial redundant elimination based on question propagation is presented [17].

6.3.1 Rank

We assume that each CFG node is numbered in topological order on CFG ignoring up-edges which go from a node to its ancestor in depth-first spanning tree. The topological number, which we call *rank*, is recursively defined as the following function $rank(v)$:

Definition 1

1. *If v is start node, $rank(v)$ is 0,*

2. *Otherwise $rank(v)$ is $max(\{rank(p) \mid p \in pred(v) \wedge (p, v) \neq \text{up-edge}\}) + 1$, where $max(S)$ denotes the maximal element of a set S.* ∎

We introduce a heuristics based on the rank to check whether an inquired expression occurs on paths from the dominator of current node v to v, without traversing the predecessors succeedingly. Let's consider CFG G_{noUp} with no up-edge. Then, we have the following lemma:

Lemma 1
Let d be a dominator of node v in G_{noUp}. In G_{noUp}, questions about e propagated to v's predecessors do not reach any occurrences of e until d, if there is no occurrence at any node x satisfying $rank(d) < rank(x) < rank(v)$.

Proof: Since G_{noUp} does not have up-edge, subgraph between d and v forms a DAG. In such a subgraph, if e occurs at node x on paths $(d...v)$, x satisfies $rank(d) < rank(x) < rank(v)$ because of property of topological order on DAG. In contraposition to that, if there is no occurrence at x satisfying $rank(d) < rank(x) < rank(v)$, questions propagated to v's predecessors cannot reach any occurrences during propagations from v to d. ∎

Next, consider CFG G_{up} with up-edges as shown by Figure 6.4(a). Since a question propagated to source node of an up-edge may reach an occurrence at x not satisfying $rank(d) < rank(x) < rank(v)$, G_{up} does not always hold lemma 1. However, considering G'_{up} obtained by ignoring up-edges from G_{up} as shown by Figure 6.4(b), it can be easily determined whether there are some destinations of the up-edges between d and v such as lemma 1. We obtain the following lemma about G_{up}:

Lemma 2 *Let d be a dominator of node v in G_{up}. G_{up} forms a DAG between d and v if v is not destination of up-edges and there is no destination u of up-edges satisfying $rank(d) < rank(u) < rank(v)$.*

Proof: In G'_{up}, subgraph between d and v forms a DAG, so if destination u of up-edge is on the DAG, u is v itself or u satisfies $rank(d) < rank(u) < rank(v)$ because of property of topological order on DAG. In contraposition to that, if there is no u satisfying the condition, u does not exist on any paths $(d...v]$, which means G_{up} corresponds to G'_{up} between d and v, and forms a DAG between them. ∎

From lemma 1 and 2, we have the following theorem for an arbitrary CFG.

Theorem 1 *Let d be a dominator of node v in CFG. the question about e propagated to v's each predecessor does not reach any occurrences of e until d under the following conditions:*

1. *v is not a destination of up-edge and there is no destination u of up-edge satisfying $rank(d) < rank(u) < rank(v)$, and*

2. *there is no occurrence of e at x satisfying $rank(d) < rank(x) < rank(v)$.*

Proof: The first condition implies that subgraph G_{sub} between d and v forms a DAG from lemma 2. Therefore, lemma 1 can be applied to G_{sub}, so from the second condition, we find that questions propagated to v's predecessors do not reach any occurrence of e until they reach d. ∎

Let's suppose that a required question is not answered at current node v. Then, if the assumption of theorem 1 holds, the question at v can be propagated to its dominator d directly. In general, there may be some exclusive subgraphs between two nodes, but note that propagation based on the above results is conservative and therefore it is also valid in such a case.

6.3.2 Sparse edges

Theorem 1 can be applied to CFG in which every loops have been collapsed into single nodes. Here, if sparse edge satisfies the condition that its source is included in loop L immediately enclosing its destination and all loops included in L have been collapsed, the sparse edges satisfy the asumption of lemma 2. Actually, we adopt such edges as sparse edges. In detail, the source s of each sparse edge is determined by type of loop L enclosing it as follows:

1. if L is reducible, s is a loop entry, and
2. if L is irreducible, s is an immediate child of common nearest dominator of loop entries.

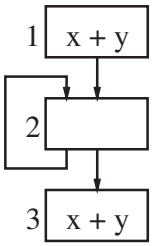
As a result, introducing these sparse edges enable propagating questions as far as possible.

However, such sparse edges are not always consistent with ranks computed by mentioned rule. For example, the ranks of node 9 and 14 in Figure 6.4 are both 7. In this case, since the rank of node 9 is not included in the region between the ranks of node 1 and 14, undesirable propagation along sparse edge (1,14) cannot be detected by the process of checking ranks. To avoid this inconsistency between sparse edges and ranks, we generate virtual edges for all pairs of sources of up-edges and successors of loop exits. For example, in Figure 6.4, edge (9,14) is a virtual edge. After that, correct ranks can be computed by only applying definition 1 to CFG modified by virtual edges.

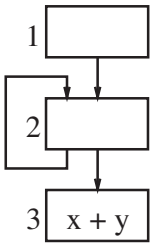
6.3.3 Partial redundancy elimination

Next, we extend question propagation to enable partial redundancy elimination (PRE). The basic idea is inserting expressions at the some predecessors of node v where the answer of required question is false, and making v's answer true while v's answer is false if the answers of all predecessors of v are false. Furthermore, to insert expressions without decreasing PRE's effectiveness, the insertion node has to satisfy the condition that the questionary expression is anticipated at the node. We check the condition by propagating required question forwardly. Since this forward question propagation is only applied when an expression is inserted, its frequency is not as often as question propagation.

However, this inserting manner may cause the problem of hoisting through loops. For example, since availability of `x+y` at node 2 in Figure 6.5(a) is true, the answer at node 1 is true and back-edge's source is also true. As well, since back-edge's source in Figure 6.5(b) is true, the questionary expression is inserted at node 1. Such hoisting through loops is not only ineffective, but is also known that it is harmful because of increasing register pressure. So we introduce *weak availability* as a sort of availability. Weak availability WAV become true if the question is propagated to the visited node where the question has not been answered yet while availability AV become false. After that, WAV and AV



(a) Availability through loops



(b) Hoisting through loops

Figure 6.5: Hoisting through loops

at node v are computed as follows:

$$WAV_v = AV_v \vee \prod_{v' \in pred(v)} WAV_{v'}$$

$$AV_v = \sum_{v' \in pred(v)} AV_{v'}$$

As shown by the equations, v's WAV become true if all predecessor's WAVs are true or some predecessor's AVs are true, so that insertion of Figure 6.5(b) is not caused. In this case, v's answer corresponds to WAV.

Finally, we show our entire algorithms in Figure 6.6, 6.7 and 6.8.

```
eqp(e,v)
1  if isOccurWithStackPush(e,v)
2     availVar[v] = top(); return(true)
3  end if
4  if visited[v]=e then
5     avVar←availVar[v]
6     if (avVar=⊥)
7        add t←⊥ to insertCpy (t is a fresh temporary)
8        push(t,false,true) // additionally push the attributes AV = false, WAV = true
9        return(true)
10    else if (avVar=⊤) return(false)
11    else push(avVar,true,true); return(true) // additionally push the attributes AV = true, WAV = true

12 if (v is the start node ∨ isMod(e, v) ∨ visited[v]≠⊥) then
13    return(false)
14 end if
15 return propagate(e,v);

propagate(e,v)
16 visited[v]←e
17 if ¬modPhi(e, v) then
18    target←shortCutTarget[v]
19    lower←rank[rankCheckTarget[v]]
20    upper = rank[v]
21    if lower>upper then swap(lower,upper) end if
22    if (¬rankCheck(e,lower,upper))
23       if (eqp(e,target)
24          replaceBottom(v,top())
25          availVar←top(); return(true)
26       else return false
27       end if
28    end if
29 end if
30 for each node p ∈ pred(v) do
31    e'←transPhi(e,p,v)
32    if (¬eqp(e',p))
33       if (postEqp(e,v))
34          add t←e to tmpInsert[p] (t is a fresh temoprary)
35          push(t,false,false)
36       else
37          pop() |pred(v)| times
38          availVar[v]← ⊤; return false
39       end if
40    end if
41 end for
42 if (createPhiAndIsItAvail(v,tmpInsert)) // compute new stack attributes AV and WAV, and φ-function
43    replaceBottom(v,top())
44    availVar[v]←top(); return(true)
45 else availVar[v]← ⊤; return(false)
46 end if
```

Figure 6.6: Algorithm of EQP

```
postEqp(e,v)
1  if isAnt[e,v]∨visited'[v]=e return(true) end if
2  if v is the end node ∨ isMod(e, v) then
3     return(false)
4  end if
5  if isOccur(e,v) then return(true)
6  else if visited'[v]≠ ⊥ then return(false)
7  end if
8  visited'[v]←e
9  for each node s ∈ succ(v) do
10    e'← transPhi'(e,v,s)
11    if(¬postEqp(e',s)) return(false) end if
12 end for
13 isAnt[e,v]←true; return(true)
```

Figure 6.7: Algorithm of forward question propagation

```
gvn()
1  for r = 0 to maxmal rank do
2     for each node b with rank r do
3        for each statement v←e in b do
4           if ¬isMod(e)∧ propagate(e,b)
5              for each d←s∈insertCpy do
6                 for each CFG node v do
7                    replace occurrence d in φ-function included in insertIn[v] with s
8                 end for
9              end for
10             for each node v do
11                insert φ-function included in insertIn[v] to entry of v
12                insert statements included in insertOut[v] to exit of v
13             end for
14             let newSrc be an available variable of top()
15             replace e of v←e with newSrc
16             do copy propagation of v←newSrc
17          end if
18       end for
19    end for
20 end for
```

Figure 6.8: Algorithm of global value numbering based on EQP

6.4 Moving Loop Invariant Expressions Out of Loops

Moving loop invariant expressions out of loops is an example of loop optimization. A loop invariant expression is an expression where all operands of the expression e: $t \leftarrow a_1$ op a_2 in a loop perform computations with any of the following characteristics. In SSA form the characteristics are as follows.

1. a_i is a constant
2. the definition of a_i is outside the loop
3. the statement defining a_i is marked as loop invariant

To find invariant expressions in loops, we need to see whether the above characteristics hold for each expression in the loop.

If expression e in a loop is a loop invariant, basic block B containing the expression must dominate all loop exit blocks to safely move that expression out of the loop [1]. Appel et al. relaxes this condition by stating that block B containing the invariant expression must dominate all loop exit blocks whose Live Out contains variable t defined by the invariant expression [3]. In our implementation, we followed Appel's condition. Figure 6.9 shows the algorithm for moving loop invariant expressions out of loops.

```
for each loop structure L in CFG do
    for each block B in L do
      find loop invariant expression e
      if expression e is not a division and so on then
        let def(e) be the variable defined by e
        let exit(e) be the exit blocks where def(e) is in Live Out
        if B ∈ dom(E) for all E ∈ exit(e) then
          move e into preheader P of L
        end if
      else
        let exit(L) be the exit blocks of L
        if B ∈ dom(E) for all E ∈ exit(L) then
          move e into preheader P of L
        end if
      end if
      (It should be so according to the dragon book)
    end for
end for
```

Figure 6.9: Algorithm for moving loop invariant expressions out of loops

When moving loop invariant expressions out of loops, basic block B including e must dominate all loop exit block exit(e) at which variable def(e), defined by invariant expression e, is in Live Out. Thus, for while loops, i.e. a loop where the entry and exit block are both the same block, it is not possible to move invariant expressions out of blocks other than the entry block. So, it is preferable to convert while loops into do-while loops, i.e. loops where the entry and exit blocks are different blocks, before optimization. Our system implements such loop structure transformation as part of the optimization process. For information on loop structure transformation, see Section 3.1.

6.5 Operator Strength Reduction and Test Replacement for Induction Variables

Reducing operator strength is a technique that improves compiler-generated code by translating certain costly computations into less expensive ones with the same meaning [13, 10]. Our system implements operator strength reduction of basic induction variables and general induction variables based on the paper by Cooper et al. [10]

The optimization of basic and general induction variables may sometimes make replacement of the end of loop test possible [13, 10]. This is also implemented in our system.

The technique (Operator Strength Reduction, hereafter OSR) proposed by Cooper et al. optimizes the SSA graph rather than handling CFG directly. The SSA graph clarifies the link from data use to its definition, which facilitates understanding of the algorithm. For details on SSA graph, see Section 6.8.

6.5.1 Induction Variables

Induction variables are variables whose values increase/decrease by a constant in each loop. Here, a constant refers to an invariant value in a loop [13]. In SSA form, the variable i shown in Figure 6.10 is a basic induction variable. In Figure 6.10, i0 is the value at the first entry into the loop, i2 is the value after going through the loop and RC is a loop invariant expression. When i is a basic induction variable, the following linear (first-degree) expression of i

$$j \leftarrow RC_1 * i + RC_2$$

is a general induction variable. Here, RC_1 and RC_2 are loop invariant expressions. OSR optimizes such basic induction variables and general induction variables.

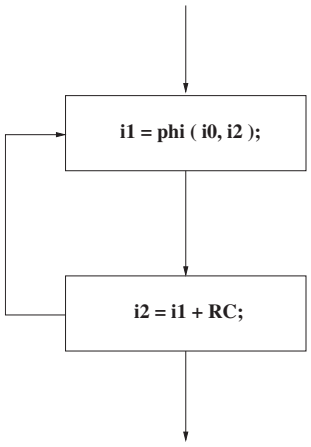


Figure 6.10: Example of induction variable

Normally, all basic induction variables take the form of strongly connected components (SCC) in an SSA graph. However, not all SCCs represent induction variables. The set of basic induction variables is a subset of SCC set. All SSCs in an SSA graph for a program contain all the basic induction variables of the program. OSR extracts SCCs from an SSA graph and examines them to find basic induction variables. If all the operators in SCC are $IV + RC$, $IV - RC$, $COPY$ operators or ϕ-functions[1], they correspond to basic induction

[1] All parameters of a ϕ-function must be in the SCC or be a loop invariant value.

variables.

Here IV indicates basic induction variable and RC loop invariant value. For details, see the paper by Cooper et al.[10]

6.5.2 Operator Strength Reduction

OSR performs depth-first search of the SSA graph to find SCCs in the graph. Figure 6.11 shows the algorithm for OSR driver and depth-first search.

```
OSR(SSAgraph)
  while there is an unvisited node n in SSAgraph do
    call DFS(n)
  done

DFS(node)
  assign DFSnum (DFS number) to node
  mark node meaning that it was visited
  node.low←node.DFSnum
  PUSH(node)
  for each "operand o of node" do
    if o has not been visited then
      call DFS(o)
      node.low←MIN(node.low,o.low)
    end if
    if (o.DFSnum < node.DFSnum ∧ o is in the stack) then
      node.low←MIN(o.DFSnum,node.low)
    end if
  end for
  if node.low = node.DFSnum then
    let SCC be empty list
    do
      x←POP()
      add x to SCC
    while (x ≠ node)
    call ProcessSCC(SCC)
  end if
```

Figure 6.11: OSR: driver and depth-first Search

After finding SCCs by depth-first search, OSR checks whether they correspond to basic induction variables. Figure 6.12 shows this algorithm. Any basic induction variable that is found is registered as IV. OSR checks whether the operator strength of SSC that do not correspond to basic induction variables can be reduced and reduces the operator strength if this is possible.

Operator strength reduction is mainly performed by the following three functions.

- *Replace*
  Rewrites the current operation with a $COPY$ from its reduced counterpart

- *Reduce*
  Inserts code to strength reduce a basic induction variable and a general induction variable and returns the SSA name of the result

- *Apply*
  Inserts an instruction to apply an opcode to two operands and returns the SSA name of the result

Figure 6.13 shows the algorithm for replacing operators and Figure 6.14 shows the algorithm for applying operators.

```
ProcessSCC(SCC)
    if SCC has a single member n then
      if n is an operator that can be replaced (Note) then
        call Replace(n,IV,RC)
      else
        n.header←NULL
      end if
    else
      call ClassifyIV(SCC)
    end if

ClassifyIV(SCC)
    for each n ∈ SCC do
      if header.RPOnum > n.block.RPOnum then
        header←n.block
      end if
    end for
    for each n ∈ SCC do
      if n's operator is not {φ, +, −, COPY} then
        SCC is not a basic induction variable
      else
        for each o ∈ operands of n do
          if o ∉ SCC ∧ ¬RegionConst(o,header) then
            SCC is not a basic induction variable
          end if
        end for
      end if
    end for
    if SCC is a basic induction variable then
      for each n ∈ SCC do
        n.header←header
      end for
    else
      for each n ∈ SCC do
        if n is an operator that can be replaced (Note) then
          call Replace(n,IV,RC)
        else
          n.header←NULL
        end if
      end for
    end if
```

(Note:) Operators that can be replaced are $x \leftarrow IV \times RC$, $x \leftarrow RC \times IV$, $x \leftarrow IV \pm RC$, $x \leftarrow RC + IV$ or $x \leftarrow IV \ll RC$.

Figure 6.12: OSR: check basic induction variable

```
RegionConst(name,header)
  if the value of name is a constant ∨ name.block ∈ sdom(header) then
    return true
  else
    return false
  end if

Replace(node,IV,RC)
  result←call Reduce(node.op,IV,RC)
  replace node with a COPY from result
  node.header←IV.header
```

Figure 6.13: OSR: replacement of operator

```
SSAname Reduce(opcode,IV,RC)
  result←call search(opcode,IV,RC)
  if result is not registered in table then
    assign new SSA name to result
    call add(opcode,IV,RC,result)
    newDef←call copyDef(IV,result)
    newDef.header←IV.header
    for each operand o ∈ newDef do
      if o.header=IV.header then
        call Reduce(opcode,o,RC)
        replace o with the result of Reduce
      else if opcode=x ∨ newDef.op=φ then
        call Apply(opcode,o,RC)
        replace o with the result of Apply
      end if
    end for
  end if
  return result

SSAname Apply(opcode,op1,op2)
  result←call search(opcode,op1,op2)
  if result is not registered in table then
    if op1.header ≠ NULL ∧ RegionConst(op2,op1.header) then
      result←call Reduce(opcode,op1,op2)
    else if op2.header ≠ NULL ∧ RegionConst(op1,op2.header) then
      result←call Reduce(opcode,op2,op1)
    else
      assign new SSA name to result
      call add(opcode,op1,op2,result)
      Determine the location where the new operation will be inserted
      Do constant folding if possible
      Insert newOper at the location determined
      newOper.header←NULL
    end if
  end if
  return result
```

Figure 6.14: OSR: application of operator

6.5.3 Test Replacement

The program translated after operator strength reduction may contain basic induction variables only in the loop-end test. Such basic induction variables can be removed by performing Linear Function Test Replacement (LFTR) [10].

To replace a test, we must find the compare instruction of the basic induction variable iv1 and a loop invariant value rc1. The compiler uses the history how operator strength of iv1 was reduced and applies the same operations in the history to rc1. For example, if operator strength of induction variable iv1 was reduced as follows,

$$iv1 \overset{-1}{\Rightarrow} iv2 \overset{\times 4}{\Rightarrow} iv3 \overset{+a}{\Rightarrow} iv4$$

then this information is used to create a loop invariant value rc, which is used in the new compare instruction.

$$rc \leftarrow ((rc1 - 1) \times 4) + a$$

This makes it possible to replace the comparison operation made up of iv1 and rc1 by iv4 and rc. The replaced induction variable iv1 is eliminated through dead code elimination (section 6.7).

6.6 Constant Propagation Considering Conditional Branch

Algorithms that perform constant propagation involve a simple algorithm that does not consider conditional branches or an algorithm that considers conditional branches. In the following, we will use an algorithm that analyzes only blocks that can be executed considering conditional branches [18, 13].

First, we will mark edges and blocks in CFG that are known to have possibility to be reached during execution. When edge e is an edge on the way towards block B, this is written as B=target(e). We use two work sets: Flow Work and SSA Work. The algorithm is shown below. A variable is evaluated as "undefined" when the definition is not executed and as "indeterminate" when its definition cannot be uniquely determined depending on flow. Constant c is the evaluation of such a variable when its value is uniquely defined as constant c.

1. Initialization:
   Flow Work = {edge from program entry node}
   SSA Work = empty set
   reachability of all edges is false
   reachability of all blocks is false
   the value of all expressions is "undefined"
2. Goto 6 if both Flow Work and SSA Work are empty.
3. If Flow Work is not empty, take edge e and remove it from Flow Work. If the reachability of e is true, do nothing. Otherwise:
   - set reachability of e to true.
   - Perform "ϕ-function evaluation" (see below) for ϕ-function in B=target(e).
   - If B's reachability is false, change it to true, and perform "expression evaluation" (see below) of all expressions in B.
   - If there is only one successor block of B (and if B does not end with a conditional branch), add the edge from B to Flow Work.
4. If SSA Work is not empty, take one definition v from there and remove it from SSA Work. For all expression exp that uses v if exp is in the reachable block, do the following.
   - if exp is a ϕ-function, perform "ϕ-function evaluation".
   - if exp is not a ϕ-function, perform its "expression evaluation".
5. Go to 2.
6. If in the ϕ-function arguments there are those from unreachable blocks, then delete them. And if such rewriting of ϕ-functions have been performed, go to 1 (We are not sure why this is necessary).

"ϕ-function evaluation":

- Obtain the values of parameters that correspond to reachable edges, perform the calculation in Figure 6.3 and perform "ϕ-function evaluation".
- If this value differs from previous value, add the definition of the ϕ-function to SSA Work.

"Expression evaluation":

- If there are any "indeterminate" operand values, let the evaluation value for the expression be "indeterminate".
- if all operands are constants, calculate expression value c, and let the evaluation value of the expression be c.
- Otherwise let the evaluation value be "undefined".
- However, if the operator is a logical "or" and one of the operands is true, let the evaluation value of the expression be true; if the operator is a logical "and", and one of the operands is false, let the evaluation value of the expression be false.
- If the above evaluation result differs from the previous value and when it is in the form "w←Expression", add them to SSA Work. When it is a conditional expression (of a conditional statement), add the edge selected by it to Flow Work. That is, if evaluation value of a conditional expression is a constant (true or false), add the edge corresponding to true or false, and if it is "indeterminate" add all edges.

Table 6.3: Calculation that determines whether ϕ-function is constant

	undefined	constant c'	indeterminate
undefined	undefined	constant c'	indeterminate
constant c	constant c	constant c(if c=c') indeterminate (if c≠c')	indeterminate
indeterminate	indeterminate	indeterminate	indeterminate

The left column and the top row are the first and second parameter of the ϕ-function.

Table 6.3 shows how the evaluation value of an expression is determined by a combination of ϕ-function parameters. For example, if the evaluation value of a_1 and a_2 of ϕ-function ϕ(a_1, a_2) are "undefined" and "constant c" respectively, the evaluation value of the ϕ-function is "constant c".

When all blocks have been processed, all code that remains "undefined" can be eliminated as dead code. Our system does that. When the optimization result indicates that there is an edge that can not be reached from the entry of CFG, the conditional branch instruction at the beginning of that edge is appropriately rewritten and the edge is eliminated from CFG.

Handling of division by 0 in constant folding

In subsection 4.2.1, we described that in our implementation, 0 or 0.0 are assigned to as initial values of SSA form variables instead of $\perp$.

However, this may cause "division by 0" at constant folding time, which actually does not occur at execution time. For example, consider the source program as the following:

```
int i;
if (...)  i = ...;
...  = 8 / i;   /* (*) */
```

We assume that the value of "i" is always set at execution time and exception by division by 0 at (*) does not acutually occur.

However, what we can know at optimization time is only that the value of "i" reaching (*) is either the value assigned to it in the if statement or the value of "i" that is not initialized. In the current implementation, values of uninitialized variables are set to 0. Therefore, if we apply the constant propagation optimization naively, the value of "i" is 0 when it is not initialized. So, during the process of constant propagation, the value of the ϕ-function just before (*) becomes temporarily 0, and the optimizer performs division (*) using that value, causing the division by 0.

Currently, as a solution to this problem, when division by 0 occurs during the constant folding, we set its value to "indeterminate". By this treatment, the algorithm in subsection 4.2.1 runs appropriately. (For a thorough solution, it will be desirable to treat the values of unitialized variables to be "indeterminate" instead of 0.)

Note in the implementation – constant folding of floating point numbers

In the current implementation of constant folding in the SSA optimization, floating point numbers are also folded. Its computation is made utilizing the constant folding API provided by the LIR module. In LIR, floating point constants are kept in the form of "double" type in Java. Therefore, if we fold a constant of "float" type of the C language, it is evaluated in the "double" type, and the result may be different from evaluating it in the "float" type, due to the difference in the precision. (Similarly, constants of "integer" type are kept in the form of "long" type in Java, and it may also have potential problems.)

6.7 Dead Code Elimination

Code that is not executed, i.e. code not reached by the control flow is dead code. Expressions like a←a are also dead code. Furthermore expressions like a←b+c are dead code if the value a is not used.

Dead code is detected as shown below.

1. Let code that is trivially "live" be $LIVE$ and other code be "dead" ($DEAD$).
2. Let code Y that defines variables used in live code X be $LIVE$.
3. Finish after doing step 2 for all $LIVE$ code. Otherwise repeat step 2.

In this system we made the following LIR be trivially "live" code.

- PROLOGUE
- EPILOGUE
- CALL
- SET to any other than register

Since this is in SSA form, for each use of a variable only one statement defines that variable. Thus no live range analysis is required for performing step 2.

Dead code elimination also eliminates dead conditional branch instructions. The conditions that render conditional branch instructions dead are as follows.

1. There is no live code that is control dependent on the conditional branch [2]

However, the above condition 1 alone would turn a program that is originally infinite loop into a program without the infinite loop by the dead code elimination. This changes the semantics of the program [3]. To prevent this, conditional branches to exit from a loop are treated as trivially live code in the current implementation.

When unnecessary conditional branches are eliminated, multiple edges to successors must be restricted to one. In our implementation, one of the edges that reach blocks containing live code is selected arbitrarily and other edges are eliminated. After edge elimination, if the destination block of the eliminated edge can not be reached from entry, that block is eliminated from CFG.

The algorithm for dead code elimination is shown in Figure 6.15 [13, 3].

[13] and [3] describe the most common algorithms for eliminating dead code in SSA form. These algorithms handle ϕ-functions in the same way as other code. They also allow change of code sequence that change the control flow under specific conditions. The conditions do not change regardless of whether the program to be optimized is in SSA form or not. However, ϕ-functions, which are specific to the SSA form, differ from other code in that they contain control flow information and data flow information. For this reason, ϕ-functions have to be processed somewhat differently from other code in the dead code elimination in the SSA form. So, we made the following two improvements.

[2] however, LIR instruction JUMP is an exception.

```
Work ← empty set
for each statement S do
  if statement S is trivially live then
    Live(S) ← true
    add S to Work
  else
    Live(S) ← false
  end if
end for
while(Work is not empty) do
  Eliminate statement S from Work, process S as described below
  for each D(define statement D of variable used in S) do
    if Live(D) = false then
      Live(D) ← true
      add D to Work
    end if
  end for
  for each block B(Block of S is control dependent on B) do
    if Live(last conditional statement of B) = false then
      Live(last conditional statement of B) ← true
      add the last conditional statement of B to Work
    end if
  end for
end while
for each statement S do
  if Live(S) = false then
    eliminate statement S
  end if
end for
```

Figure 6.15: Dead code elimination algorithm
Note: This algorithm handles ϕ-functions in the same way as other code.

1. **The conditional branch instruction on which the block associated with a ϕ-function source resource is control dependent is not deleted**
   For example, see Figure 6.16 (a). According to condition 1 above, since block P is empty the last conditional statement of B, on which P is control dependent, is eliminated by dead code elimination. This is all right only if the ϕ-function of S is not necessary (see section 6.14), but there is always the risk that it may change the semantics of the program.

2. **When variables with different SSA variable names but from the same predecessor are input to ϕ-function source resources, the conditional statement in the predecessor is not eliminated.**
   For example, see Figure 6.16 (b). According to condition 1 above, since block S is not control dependent on block P, the last conditional statement in P is eliminated as dead code. This obviously changes the semantics of the program. Using the above method 1 to prevent the change of the program semantics is not helpful: the last conditional branch P is eliminated since P $\in$ pred(S) is not control dependent on P itself.

Figure 6.17 shows an improved dead code elimination algorithm. Bold type in the algorithm indicates improvements.

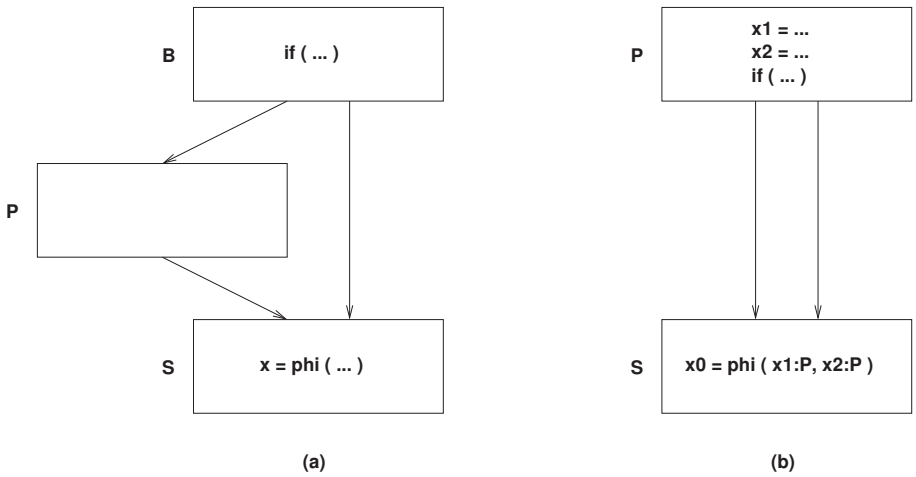


Figure 6.16: Example of dead code elimination that carries risks

```
Work ← empty set
for each statement S do
  if statement S is trivially live then
    Live(S) ← true
    add S to Work
  else
    Live(S) ← false
  end if
end for
while (Work is not empty) do
  eliminate statement S from Work and do the following for S
  for each D (define statement D of variable used in S) do
    if Live(D) = false then
      Live(D) ← true
      add D to Work
    end if
  end for
  if S is a φ-function then
    /* even if a block associated with a φ-function source resource is empty
       the conditional statement on which the block is control dependent
       should not be eliminated */
    for each block P ∈ pred(block of S) do
      for each block B (P is control dependent on B) do
        if Live(last conditional statement of B) = false then
          Live(last conditional statement of B) ← true
          add the last conditional statement of B to Work
        end if
      end for
    end for
    /* when variables with different SSA variable names but from the
       same predecessor are input to φ-function source resources,
       the conditional statement in the predecessor should not be
       eliminated. */
    if (the names of φ-function source resources differ,
        and they are associated with the same block P ∧
        Live(last conditional statement of P) = false) then
      Live(last conditional statement of P) ← true
      add the last conditional statement of P to Work
    end if
  end if
  for each block B (block of S is control dependent on B) do
    if Live(last conditional statement of B) = false then
      Live(last conditional statement of B) ← true
      add the last conditional statement of B to Work
    end if
  end for
end while
for each statement S do
  if Live(S) = false then
    eliminate statement S
  end if
end for
```

Figure 6.17: Improved dead code elimination algorithm
Note: Bold text indicates improvements.

6.8 Algorithms for Constructing SSA Graph and for Finding Equality of Variables

This section describes SSA graph (or value graph) which is a data structure that focuses on the flow of values of variables in the SSA form. Nodes in an SSA graph represent either a value or an operation. The label of the variable name which is defined by the value or operation of that node is attached to each node.

Edges are placed from nodes that represent operations to the nodes that define the variables used by the operations [2, 10]. As an example, Figure 6.19 shows the SSA graph obtained from the program in Figure 6.18.

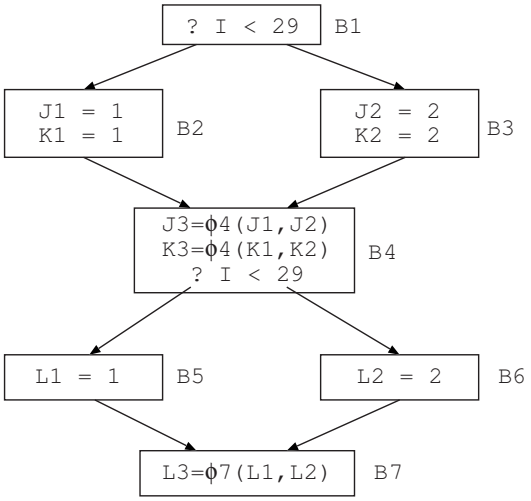


Figure 6.18: Example of program in SSA form

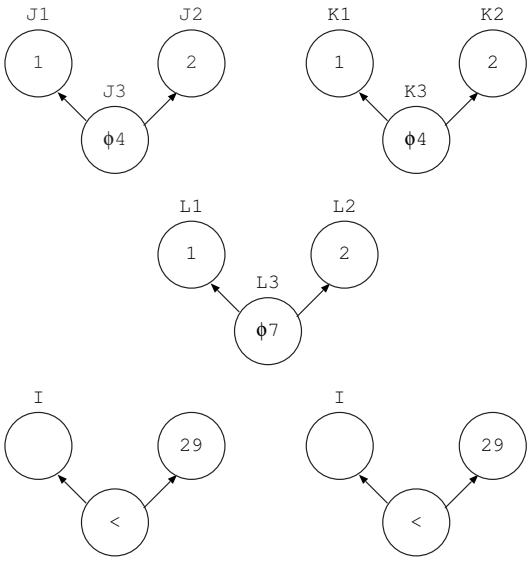


Figure 6.19: SSA graph obtained from program in Figure 6.18

The following sections describe the algorithm for building the SSA graph from LIR and the algorithm for checking the equality of values using the SSA graph.

6.8.1 Algorithm for Constructing SSA Graph from LIR

To build the SSA graph from LIR, read in each LIR instruction and extract the part of value or operation in LIR and convert them to nodes in the SSA graph. In operations that are assignment statements, use the left-hand side variable names as the label of the node of the operation in the right-hand side. Care must be taken in the handling of MEM in building an SSA graph from LIR. MEMs in LIR have two meanings: load and store. MEM used in the first parameter of SET and the third parameter of CALL mean store. To translate this kind of MEM to an SSA graph, two edges should be put: one from the MEM node to the node computing the value to be assigned; the other from the MEM node to the node of the destination address. A MEM other than those described above means load. In this case an edge is put from the MEM node to the node where the value is used.

In the SSA graph, one operation node represents one operation. For example, the expression

$$x1 \leftarrow y1 * 4 + z1$$

requires two nodes, one node representing '$*$' and one node representing '$+$'. In optimizations using the SSA graph, it is convenient to explicitly assign labels of temporary variable names when no variable name labels are assigned to operation nodes. For this purpose, our system decomposes the above expression as shown below.

$$\begin{aligned} t0 &\leftarrow y1 * 4 \\ x1 &\leftarrow t0 + z1 \end{aligned}$$

t0 is a temporary variable.

Let us use the program in Figure 6.20 as an example. The program in Figure 6.20 is

```
void func(int a[]){
  int i,ans[100];

  for(i=0;i<100;i++){
    ans[i]=a[i]+i;
  }
}
```

Figure 6.20: Building SSA graph: example program (C language form)

translated into LIR instructions in Fig. 6.21. Translating instructions shown in Figure 6.21 to SSA graph produces the result shown in Figure 6.22. "def" in the ellipse in the figure means the "label", described above.

```
#1 Basic Block (.L1):
  (PROLOGUE (0 0) (REG I32 "a.1%_1"))
  (JUMP (LABEL I32 ".L2" $1))

#2 Basic Block (.L2):
  (JUMP (LABEL I32 ".L3" $2))

#3 Basic Block (.L3):
  (PHI I32 (REG I32 "i.2%_2")
           ((INTCONST I32 0) (LABEL I32 ".L2" $7) (LABEL I32 ".L3" $2))
           ((REG I32 "i.2%_3") (LABEL I32 ".L5" $8) (LABEL I32 ".L3" $6)))
  (JUMPC (TSTLTS I32 (REG I32 "i.2%_2") (INTCONST I32 100))
         (LABEL I32 ".L4" $3)
         (LABEL I32 ".L6" $4))

#4 Basic Block (.L4):
  (SET I32 (MEM I32 (ADD I32 (FRAME I32 "ans.3")
                             (MUL I32 (REG I32 "i.2%_2")
                                      (INTCONST I32 4))))
           (ADD I32 (MEM I32 (ADD I32 (REG I32 "a.1%_1")
                                      (MUL I32 (INTCONST I32 4)
                                               (REG I32 "i.2%_2"))))
                    (REG I32 "i.2%_2")))
  (JUMP (LABEL I32 ".L5" $5))

#5 Basic Block (.L5):
  (SET I32 (REG I32 "i.2%_3")
           (ADD I32 (REG I32 "i.2%_2")
                    (INTCONST I32 1)))
  (JUMP (LABEL I32 ".L3" $6))

#6 Basic Block (.L6):
  (EPILOGUE (0 0))
```

Figure 6.21: Building SSA graph: example program (LIR form)

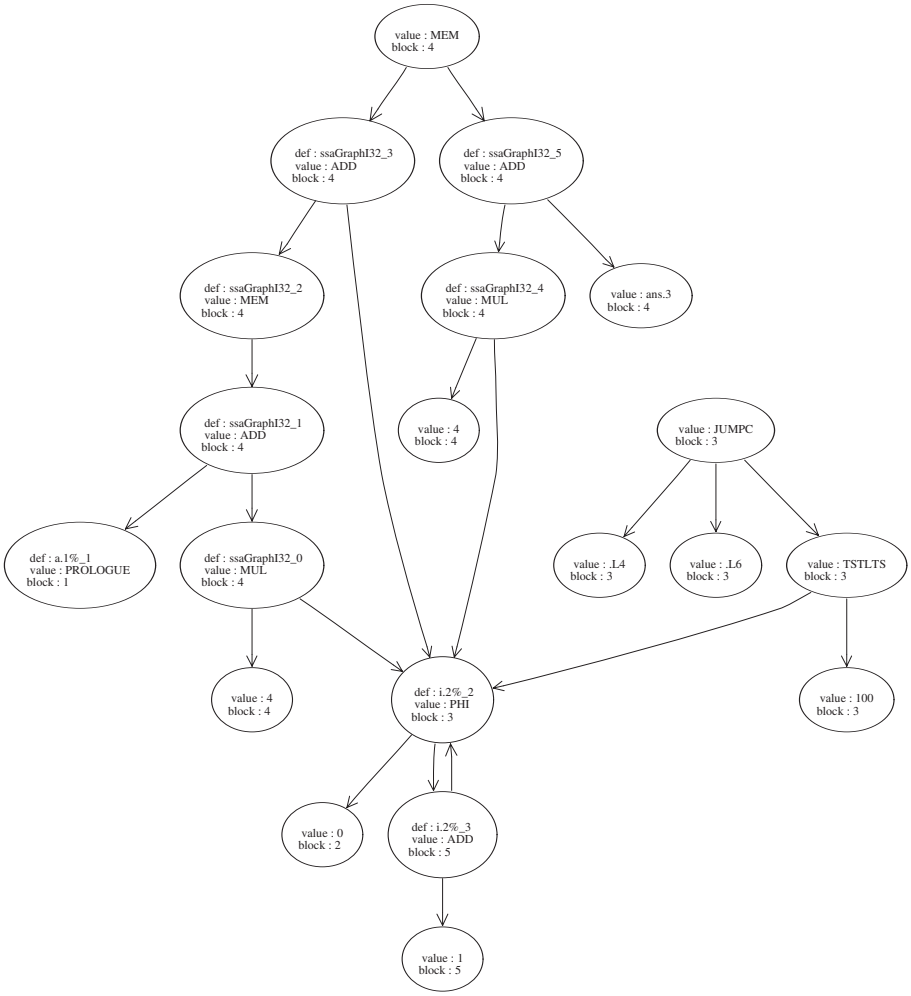


Figure 6.22: SSA graph built by translating from LIR in Figure 6.21

Shown in Figure 6.23 is the LIR which is back-translated from SSA graph in Figure 6.22. As described above, expressions that contain multiple operations are decomposed.

```
#1 Basic Block (.L1):
  (PROLOGUE (0 0) (REG I32 "a.1%_1"))
  (JUMP (LABEL I32 ".L2" $1))

#2 Basic Block (.L2):
  (JUMP (LABEL I32 ".L3" $2))

#3 Basic Block (.L3):
  (PHI I32 (REG I32 "i.2%_2")
           ((INTCONST I32 0) (LABEL I32 ".L2" $7) (LABEL I32 ".L3" $2))
           ((REG I32 "i.2%_3") (LABEL I32 ".L5" $8) (LABEL I32 ".L3" $6)))
  (JUMPC (TSTLTS I32 (REG I32 "i.2%_2") (INTCONST I32 100))
         (LABEL I32 ".L4" $3)
         (LABEL I32 ".L6" $4))

#4 Basic Block (.L4):
  (SET I32 (REG I32 "ssaGraphI32_0")
           (MUL I32 (INTCONST I32 4)
                    (REG I32 "i.2%_2")))
  (SET I32 (REG I32 "ssaGraphI32_1")
           (ADD I32 (REG I32 "a.1%_1")
                    (REG I32 "ssaGraphI32_0")))
  (SET I32 (REG I32 "ssaGraphI32_2")
           (MEM I32 (REG I32 "ssaGraphI32_1")))
  (SET I32 (REG I32 "ssaGraphI32_3")
           (ADD I32 (REG I32 "ssaGraphI32_2")
                    (REG I32 "i.2%_2")))
  (SET I32 (REG I32 "ssaGraphI32_4")
           (MUL I32 (REG I32 "i.2%_2")
                    (INTCONST I32 4)))
  (SET I32 (REG I32 "ssaGraphI32_5")
           (ADD I32 (FRAME I32 "ans.3")
                    (REG I32 "ssaGraphI32_4")))
  (SET I32 (MEM I32 (REG I32 "ssaGraphI32_5"))
           (REG I32 "ssaGraphI32_3"))
  (JUMP (LABEL I32 ".L5" $5))

#5 Basic Block (.L5):
  (SET I32 (REG I32 "i.2%_3")
           (ADD I32 (REG I32 "i.2%_2")
                    (INTCONST I32 1)))
  (JUMP (LABEL I32 ".L3" $6))

#6 Basic Block (.L6):
  (EPILOGUE (0 0))
```

Figure 6.23: LIR back-translated from SSA graph shown in Figure 6.22

6.8.2 Algorithm for Finding Equality of Variables

The algorithm for eliminating common subexpressions in the SSA form shown in Figure 6.2 finds expressions with the same values. It uses a correspondence table representing the relation between expressions and variable names and a correspondence table representing the relation between variable names and their copies shown in Table 6.1 and Table 6.2. This simple algorithm is not powerful enough to analyze loops. Many methods for finding a wide range of equality of expressions in SSA form have been proposed. Our system implements the algorithm for finding equality of variables in SSA graphs (value graphs) proposed by Alpern et al.

The following method for finding equality of variables is used. First, expressions with the same operators may have the same value and are therefore placed in the same set. Then for expressions in the same set, if the values of their operands or parameters differ from others in the same set, they are partitioned and placed in a different set. This partitioning continues until no more partitioning can be performed. Finally, expressions in the same set have all the same value. Constants with the same value are placed in one set. The left-hand side and the right-hand side of assignment statements are treated as identical and grouped by the left-hand side element. ϕ-functions in different blocks are handled as separate functions. The reason for this is that if ϕ-functions in B4 and B7 in the SSA form program shown in Figure 6.18 are handled as the same ϕ-function, L3 will be assumed to be the same as J3 and K3, which is false.

As an example, consider the program in Figure 6.24. The program translated into SSA form in shown in Figure 6.25. The conditional expressions have been omitted for the sake of brevity.

```
a=1;
b=1;
do{
  if(...){
    a=a+1;
    b=b+1;
  }
  else{
    a=a+2;
    b=b+2;
  }
}while(...);
```

Figure 6.24: Equality of variables: example program

First the following sets are created.

$$
\begin{aligned}
B[1] &= \{1, a0, b0\} \\
B[2] &= \{2\} \\
B[3] &= \{a2, b2\}\ function\ \phi 2 \\
B[4] &= \{a3, b3, a4, b4\}\ add \\
B[5] &= \{a1, b1\}\ function\ \phi 5
\end{aligned}
$$

What we want to find is the coarsest (weakest) partition such that for example, a+b and c+d are placed in the same set only if a and c, b and d are in the same set. Conversely, if a and c are already in separate sets, a+b and c+d must also be placed in different sets.

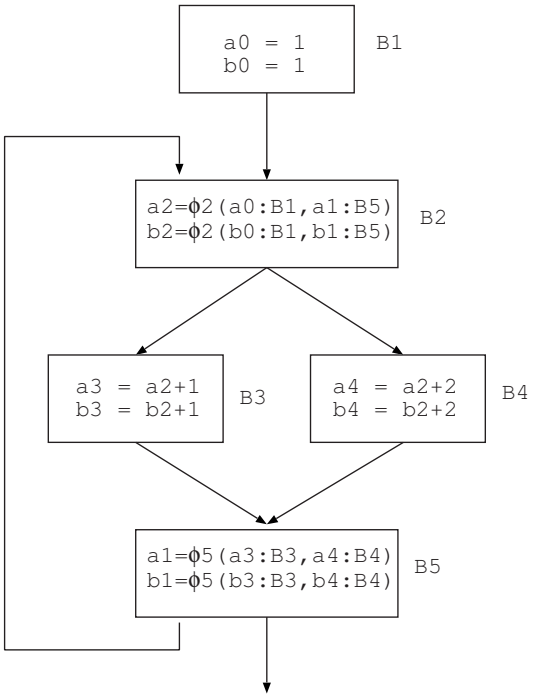


Figure 6.25: Equality of variables: example program (CFG) translated into SSA form

Consider a set of expressions H with the m-th operand that is an element of a set B[i]. If H has common elements with existing set B[j] and B[j] is not part of H, then there is an expression in B[j] whose m-th operand is an element other than B[i]. This means that B[j] need to be further partitioned. The algorithm for doing this is shown in Figure 6.27. The SSA graph is used to obtain INVERSE. The SSA graph for Figure 6.25 is shown in Figure 6.26.

The following result is obtained when algorithm in Figure 6.27 is applied to the example program. Let us see B[1]. The set of expressions whose first operand is in B[1] (its value is 1) is

$$INVERSE = \{a2, b2\} = B[3]$$

This does not satisfy the partitioning conditions. However, the set of expressions whose second operand is in B[1] is

$$INVERSE = \{a3, b3\}$$

This has common elements with B[4], but B[4] is not included in INVERSE. Then

$$B[6] = \{a3, b3\}$$

is newly created, this results in

$$B[4] = B[4] - B[6] = \{a4, b4\}, WAITING = \{2, 3, 4, 5, 6\}$$

After that there are no more elements in WAITING that meet the condition for partition-

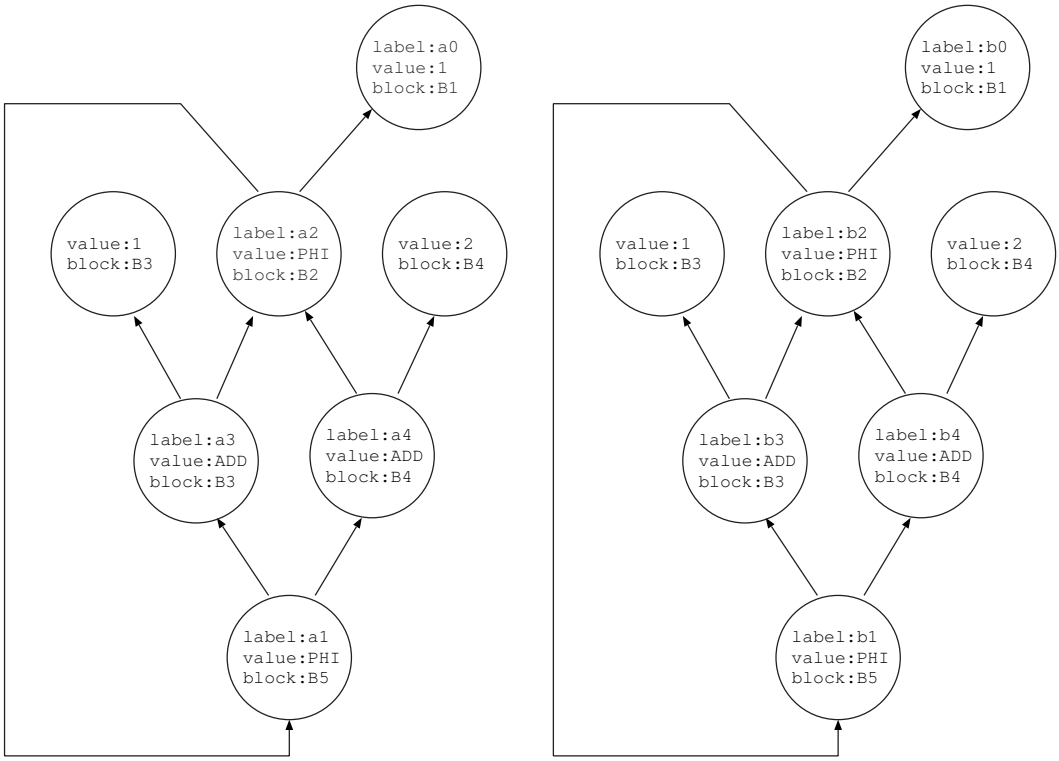


Figure 6.26: Equality of variables: SSA graph for Figure 6.25

ing. Thus the final result is as follows.

$$
\begin{aligned}
B[1] &= \{1, a0, b0\} \\
B[2] &= \{2\} \\
B[3] &= \{a2, b2\} \\
B[4] &= \{a4, b4\} \\
B[5] &= \{a1, b1\} \\
B[6] &= \{a3, b3\}
\end{aligned}
$$

All elements in the finally obtained set B[i] have the same value (at the time when these assignment statements are executed). Two variables in the same set are said to be *congruent*.

```
WAITING ← {1,2, ···, p} /* p is the number of the initial set */
q ← p
while WAITING ≠ { } do
   select one integer i from WAITING and remove it from WAITING
   for each m (1≤m≤k) do /* k is the number of operands of expression */
      INVERSE ← { }
      for each x in B[i] do
         INVERSE ← INVERSE ∪ F^-1[m,x]
         /* F^-1[m,x] is a set of expressions whose m-th operand is an element
              X in B[i] */
      end for
      for each j such that B[j] ∩ INVERSE ≠ { } ∧ B[j] ⊈ INVERSE do
         q ← q+1
         create new set B[q]
         B[q] ← B[j] ∩ INVERSE
         B[j] ← B[j]-B[q]
         if j ∈ WAITING then
            add q to WAITING
         else if |B[j]|≤|B[q]| then
            add j to WAITING
         else
            add q to WAITING
         end if
      end for
   end for
end while
```

Figure 6.27: Equality of variables: partitioning algorithm

6.9 Dividing into three-address code

In a high-level programming language such as C, there may be more than one operators in the right-hand side expression of an assignment. Therefore, the input to the compiler may be an expression like $x \leftarrow a + b * c - d$. Figure 6.28 shows the tree structure of this expression. For some optimizers, however, it is suitable that each expression has only

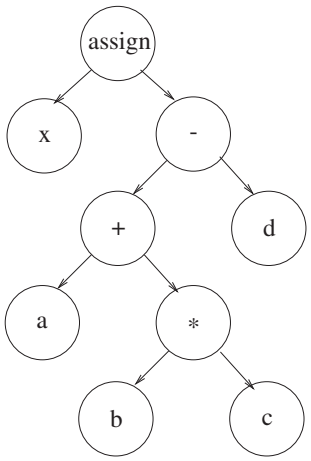


Figure 6.28: A tree-structured expression

one operator, which is called "three-address code". The SSA module can divide each expression into three-address code.

This transformation is very simple. The transformer divides an expression if an operand is not a leaf node of the tree structure. For example, the expression shown in Figure 6.28 is divided at the dashed lines, which is shown in Figure 6.29.

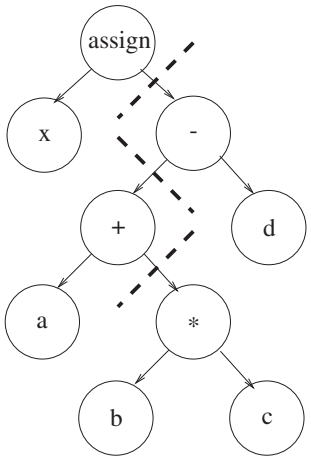


Figure 6.29: The position for dividing

For dividing expressions, we need some temporary variables to which the result of internal operations are stored. In the dividing process shown in Figure 6.29, the transformer uses $tmp1, tmp2, tmp3$ as temporary variables and transforms the expression into the expression shown in Figure 6.30. The dashed arrows in Figure 6.30 represent edges from definition to use of temporary variables.

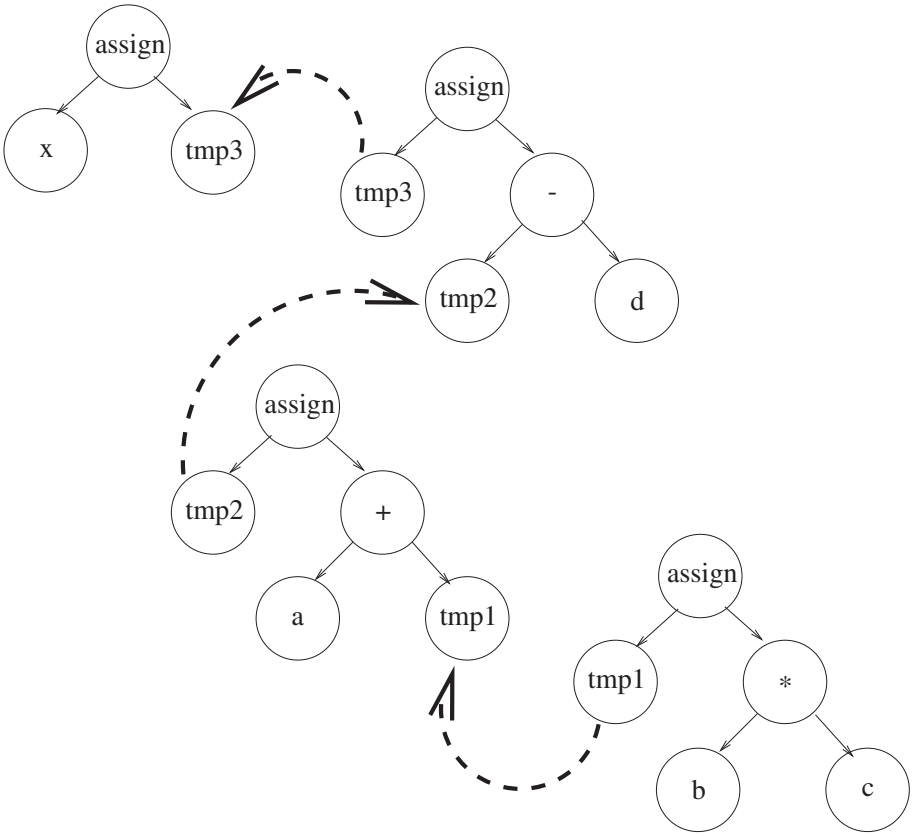


Figure 6.30: Sample of dividing an expression

Thus, the transformer in the SSA module divides the expression $x \leftarrow a + b * c - d$ into expressions listed below:

$$\begin{aligned} tmp1 &\leftarrow b \times c \\ tmp2 &\leftarrow a + tmp1 \\ tmp3 &\leftarrow tmp2 - d \\ x &\leftarrow tmp3 \end{aligned}$$

In the current implementation, the transformer generate $x \leftarrow tmp2 - d$ instead of $tmp3 \leftarrow tmp2 - d$ and $x \leftarrow tmp3$. This is for avoiding generation on of copy statements.

6.10 Elimination of Empty Blocks

Code optimization in CFG may sometimes result in the total elimination of code in a basic block. Even if control reaches such a basic block from a predecessor, the control is just passed to its successor and there is no effect on the program. Thus if this CFG structure is translated to assembler code this will increase unconditional jumps, which may eventually raise the cost of the program.

Therefore, we implemented an optimization which eliminates empty basic blocks. An empty basic block is defined as follows.

Empty basic block A basic block that does not contain any other code than jump instructions.

Applied to LIR the above definition means a basic block that does not contain any instructions other than JUMP, JUMPC and JUMPN. Here instructions include ϕ-functions. However, it is difficult to safely eliminate all empty blocks. So we made our system eliminate only empty blocks that meet the following conditions.

Condition 1 An empty basic block E that satisfies ($|\text{succ(E)}| = 1 \wedge |\text{pred(E)}| = 1$) and ($|\text{succ(pred(E))}| = 1 \wedge |\text{pred(succ(E))}| = 1$)

Condition 1 is shown in Figure 6.31 (a).

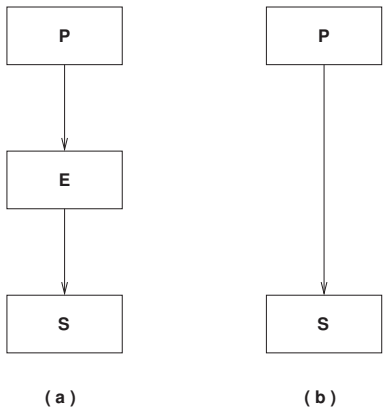


Figure 6.31: Eliminating empty blocks : Condition 1

To eliminate block E, which meets condition 1, add edge $P \longrightarrow S$ from P $\in$ pred(E) to S $\in$ succ(E) and eliminate E from CFG. Figure 6.31 (b) shows the program after elimination of block E.

6.11 Concatenate Basic Blocks

In the current implementation, COINS generates the control flow graph shown in the middle of Figure 6.32 from the program listed in the left of Figure 6.32.

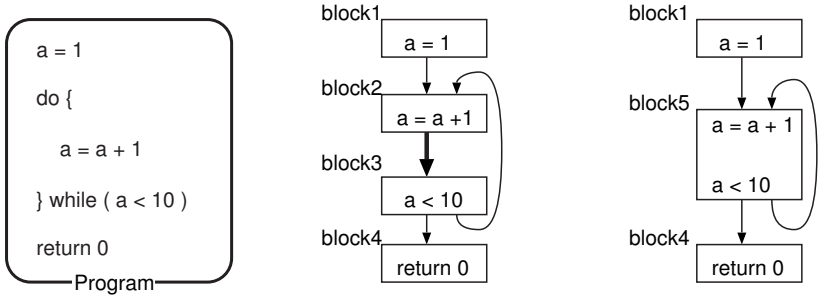


Figure 6.32: Redundant Edges

In the control flow graph shown in the middle of Figure 6.32, the edge between `block2` and `block3` is redundant, and the back end may generate a redundant branch instruction. If there is a basic block which has only one successor, like `block2`, and its successor block which has only one predecessor, like `block3`, the edge between these basic blocks is redundant, and the optimizer can eliminate them.

We implement the optimizer which concatenates basic blocks like `block2` and `block3`. The right of Figure 6.32 shows `block5` by concatenating `block2` and `block3`, and removing the redundant edge.

In the current implementation, the optimizer does not generate a new basic block `block5` but merges the instructions in the original basic blocks and puts them into one of the original basic block.

6.12 Elimination of Critical Edges

Translating SSA form into normal form or certain optimization often involve the choice whether or not critical edges [5, 13] are to be eliminated. A critical edge is an edge $B1 \longrightarrow B2$ from basic block B1 to basic block B2 where |succ(B1)| > 1 and |pred(B2)| > 1. Figure 6.33 shows an example of critical edges. The example in the figure shows critical edge $P1 \longrightarrow S2$.

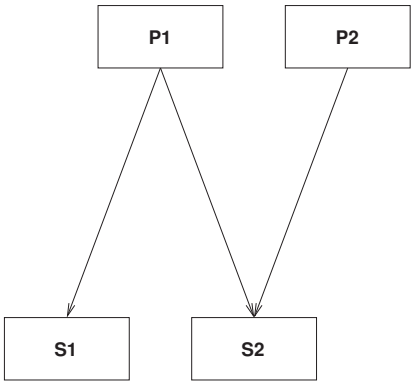


Figure 6.33: Critical edge

A critical edge can be eliminated by inserting a new basic block in that edge (edge split)[13]. For example, in the example shown in Figure 6.33, creating a new basic block N1 for critical edge $P1 \longrightarrow S2$ and changing it to $P1 \longrightarrow N1 \longrightarrow S2$ makes it possible to eliminate the critical edge. The result of elimination of critical edge in Figure 6.33 is shown in Fig 6.34.

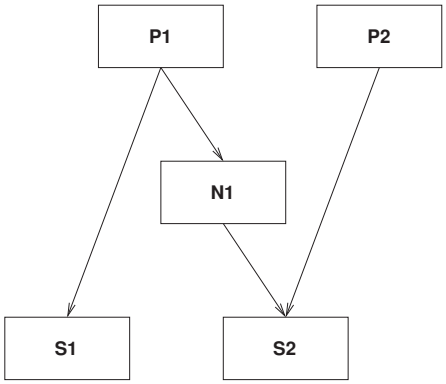


Figure 6.34: Elimination of critical edge

This method for eliminating critical edges is performed in SSA form. Changing the control flow in SSA form such as adding a basic block make handling of ϕ-functions more complex. However, our implementation ensures that the ϕ-functions are correctly rewritten and that the SSA form is retained.

6.13 Global Reassociation

We use a technique called global reassociation [5] to address the code shape problems, that is, the problem of the shape of code which is subject to the optimization. For example, $x1$ and $y1$ in the expressions listed below are mathematically equal.

$$x1 \leftarrow a1 + b1 + c1 \tag{6.1}$$

$$y1 \leftarrow b1 + c1 + a1 \tag{6.2}$$

However, it is difficult for intermediate representations of compilers such as LIR to detect whether the right hand side of (6.1) is equal to the right hand side of (6.2) because the intermediate representation are usually tree-structured (See Figure 6.35). Therefore,

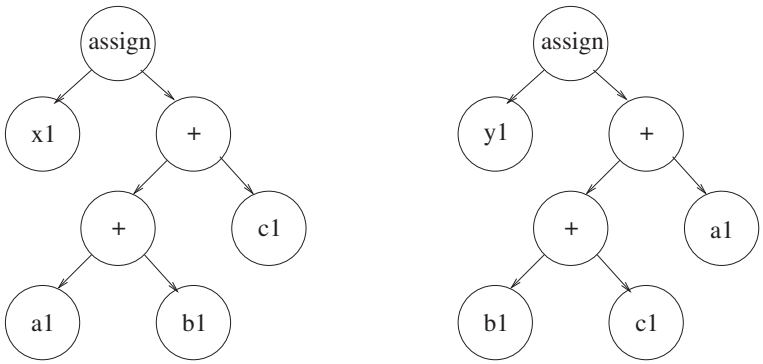


Figure 6.35: A sample of tree-structured expression

the global reassociation uses algebraic properties of arithmetic for the intermediate representations to rearrange them. It uses commutativity, associativity, and distributivity for tree-structured codes and formalize them. It is useful to expose more common subexpressions and loop-invariant expressions.

The global reassociation on this SSA module has 2 steps.

1. Compute ranks of every expression
2. Reassociate and sort expressions based on their operands' ranks

Briggs and Cooper proposed the global reassociation which had 3 steps [4], on which our approach is based. The 2nd step of theirs was "Propagate expressions forward to their uses". We don't use this step because this transformation can not keep the SSA form.

6.13.1 Computing Ranks

To guide reassociation, the SSA module assigns a rank to each expression and subexpression. Intuitively, we want constants and loop-invariant expressions to have lower ranks than loop-variant expressions.

In practice, we compute ranks on the SSA form of the routine during a reverse-postorder traversal of the CFG. Each block is given a rank as it is visited, where the first block visited is given rank 1, the second block is given rank 2, and so on. Each expression in a block is ranked using three rules:

1. A constant receives rank 0.

2. The result of a ϕ node receives the rank of the block, as do any variables modified by procedure calls. This includes the result of a load instruction.

3. An expression receives a rank equal to its highest-ranked operand.

6.13.2 Sorting Expressions

After assigning ranks to each expression, we reorder the operands in every expression. This is called "sorting". We use commutativity and associativity to do it. In order to get more opportunities for reassociation, we rewrite certain operations. We rewrite expressions of the form $x - y + z$ as $x + (-y) + z$, since addition is associative and subtraction is not. On the other hand, we avoid rewriting x/y as $x \times (1/y)$ to avoid introducing precision problems.

Sorting Expression Based on Rank

In sorting expressions based on rank, we rewrite the operands by rank so that the low-ranked operands are placed at the left side. For example, if there is a subexpression $x+y+z$ and the ranks of each operands are $3, 1, 1$, respectively, we rewrite this subexpression into $y + z + x$ or $z + y + x$. Using the optimizer "dividing into three-address code" described in Section 6.9, we can get the expresion like follows:

$$\begin{aligned} temp &\leftarrow y + z \ (or \ z + y) \\ &\leftarrow temp + x \end{aligned}$$

If y and z are loop-invariant variables, then the expression $temp \leftarrow y + z$ (or $z + y$) can be hoisted. Furthermore, since constants are given rank 0, all the constant operands in a sum will be sorted together. For example, the expression $1 + y + 2$ becomes $1 + 2 + y$. Constant propagation can easily turn the reordered expressions into $3 + y$ [3].

In our implementation, the operators of LIR listed below are the target of this sorting (SUB is not in the target).

- ADD
- MUL
- BAND
- BOR
- BXOR

Alphabetical Sorting

Sorting the operands of expressions is useful for optimizers such as Common Subexpression Elimination because these optimizers usually assume the lexical equality of operands. However, the sorting based on rank does not rewrite the operands which have the same rank. They can't understand that $a1 + b2$ equals $b2 + a1$, if $a1$ and $b2$ have the same rank. Therefore, it is desirable to sort such expressions.

In the SSA module, we implemented another method of sorting. The target of this method is the operands which have the same rank, and the corresponding operators are one of the following. It rewrites them in the alphabetical order.

[3] The constant propagation implemented in COINS LIR can translate $1 + y + 2$ into $3 + y$.

- ADD
- MUL
- BAND
- BOR
- BXOR

6.13.3 Application of Distributive Law

After sorting expressions, we look for opportunities to distribute multiplication over addition; that is, we rewrite expressions of the form $w \times (x + y + z)$ as $w \times x + w \times y + w \times z$. This distribution is not always profitable, so we again use rank as a guide. In our current implementation, we distribute a low-ranked multiplier over a higher-ranked sum.

For example, if we have an expression $a + b \times ((c + d) + e)$, where a, b, c and d have rank 1 and e has rank 2, we would distribute partially, giving $a + b \times (c + d) + b \times e$. This allows optimizer to hoist $a + b \times (c + d)$ even if $b \times e$ cannot be hoisted. Note that complete distribution would result in extra multiplications without allowing any additional code motion. It is important to re-sort sums after distribution.

When we apply distributive law to the expression $a \times (b + c)$, we get $(a \times b) + (a \times c)$ as a result. Unfortunately, this translation may lead a wrong result of calculation with overflow even if it is mathematically correct. For example, if b is set the value INT_MAX and c is set the value $-(INT\_MAX - 1)$, the result of calculation of $a \times (b + c)$ is not equal to that of $(a \times b) + (a \times c)$. Notice that our implementation includes this problem.

6.14 Removing Useless ϕ-instructions

Copy propagation and copy folding (copy propagation is an optimization pass described in section 6.1; copy folding is a process performed during translation into SSA form described in section 4.2.2) and other optimizations often result in generating useless ϕ-instructions.

The optimization procedure described in this section eliminates useless ϕ-instructions produced by other optimizations. Useless ϕ-instructions are [6, 8]:

1. ϕ-instructions whose resources are all the same
2. ϕ-instructions whose source resources are all the same [4]
3. ϕ-instructions whose source resource contains the destination resource and resources other than the destination resource in source resources are of only one kind [5]

To be precise, in conditions 2 and 3, we must check whether the source resources are $\perp$. However, the current implementation does not check whether a value is $\perp$. This is because variables having value $\perp$ created during SSA translation, i.e. the initial values pushed onto the stack for use in the renaming variable phase, are initialized to 0 at CFG entry and their values are not indeterminate.

Methods for removing useless ϕ-instructions described in 1 to 3 are as follows [8, 6].

- As for 1, eliminate the ϕ-instructions.
- As for 2, eliminate the ϕ-instructions and replace subsequently used destination resources of these ϕ-instructions with source resources. For example, in handling y$\leftarrow$ ϕ(x,x,x), the same semantics is retained by considering the ϕ-instruction as y$\leftarrow$x and performing copy propagation.
- As for 3, eliminate the ϕ-instructions and replace subsequently used destination resources of these ϕ-instructions with the source resource that is not the destination resource. For example, in handling y$\leftarrow$ ϕ(x,x,y), the same semantics is retained by considering the ϕ-instruction as y$\leftarrow$x and performing copy propagation.

[4] For example: y$\leftarrow$ ϕ(x,x,x)
[5] For example: y$\leftarrow$ ϕ(x,x,y)

6.15 Restriction of the implementation

- Volatile variables

  The optimization of programs containing volatile variables has not been implemented yet, although a framework for them is prepared.

Chapter 7

Simple Alias Analysis

The object of our optimization in SSA form are virtual registers. We realize SSA form by letting assignments to one register be statically single. Virtual registers are registers of infinite capacity to which subprogram variables other than variables that may have any of the aliases listed below are allocated.

- global variable
- (local variable, but) variable declared as "static"
- operand of & operator
- array, array element
- structure, structure element
- union, union element
- extern variable (not local variable)

Since variables assigned to a virtual register obviously have no aliases, they are easy to optimize in the SSA form. By contrast, common C programs may include array variables and variables pointed by pointers which may have multiple aliases. It is desirable that further optimization process such kinds of variables. However, optimization in SSA form becomes difficult when aliases are present. It requires more accurate alias analysis to allocate more variables without aliases to virtual registers and widening the applicability of optimization.

As a first approximation in this direction, our system has implemented a method of handling all program memory reference as one large object. We call it a simple alias analysis. This makes it possible to widen optimization range to include MEM expressions of LIR. Thus simple alias analysis and common subexpression elimination applied to programs like the one in Figure 7.1(a) is expected to produce the result shown at (b).

This simple alias analysis method regards "write to memory" as changes of all values in the memory space regardless of the written variable names. For example, consider expressions (1) to (3) below.

```
(SET I32 (MEM I32 (FRAME I32 "x") (CONST I32 0)) - (1)
(SET I32 (MEM I32 (FRAME I32 "y") (CONST I32 0)) - (2)
(SET I32 (REG I32 r1) (MEM I32 (FRAME I32 "x"))) - (3)
```

```
a[i] = 1;                    a[i] = 1;
x = a[i];                    x = a[i];
b = a[i] + 1;                b = x + 1;
c = b * a[i] + 1;            c = b * x + 1;
    (a)                          (b)
```

Figure 7.1: Example showing effect of simple alias analysis and common subexpression elimination

The value 0 is assigned to x and y in expressions (1) and (2). When the value 0 is assigned to y in expression (2), this does not corrupt the value of x. However, this simple alias analysis method assumes that all values in memory has been rewritten when a write operation is made to memory. So it cannot assume that x has value 0 in expression (3).

This method applies the same algorithm as used for SSA translation to MEM expressions and virtually sets the second operand of the MEM expression. The second operand is INTCONST whose value is a consecutive number starting from 0. The above example is translated as shown below.

Thus if the second operand of the MEM expression is the same value, this is an assurance that the memory content has not changed.

```
(SET I32 (MEM I32 (FRAME I32 "x") (INTCONST 0)) (CONST I32 0)) - (1)'
(SET I32 (MEM I32 (FRAME I32 "y") (INTCONST 1)) (CONST I32 0)) - (2)'
(SET I32 (REG I32 r1) (MEM I32 (FRAME I32 "x") (INTCONST 1)))  - (3)'
```

If a basic block with memory assignments has dominance frontiers, multiple definitions to memory must be unified, like in translation of virtual registers into SSA form. In translation to SSA form of virtual registers, a virtual instruction ϕ-function was used. However, this cannot be used for memory. Instead the value of the second operand (consecutive number) is incremented by one at the dominance frontiers to unify definitions.

In the current implementation, the result of the simple alias analysis of memory references is represented by adding information to LIR (MEM expression) that does memory references. That is, it is realized without addition of new LIR or extension. Therefore, we need not to change the process of optimization depending on whether or not the simple alias analysis has been performed [1].

[1] It means that this ensures the existing optimization can still be run after performing the simple alias analysis. Addition to the existing optimization process is necessary to perform optimization using the result of the simple alias analysis of memory.

Chapter 8

Translators from LIR to C Program

The output of our optimizer is in LIR form. The following method can be considered to test whether or not the optimized code in SSA form is correct.

1. make a visual check of the translated LIR
2. submit the translated LIR to the code generator of the COINS platform to obtain the executable code and execute it

Method 2 is an easy test since it enables comparison of result output of executed code of relatively reliable compilers such as gcc. However, this method cannot check whether the code translation performed by our optimization is a correct translation that satisfies the objective of that optimization. Method 1 can check whether the optimization result produced by the system is a translation that satisfies the objective of the optimization.

From the viewpoint of the developer of compiler optimizers, method 1 seems to be the best method for checking whether the optimized code is correct. However, it is difficult to check optimized code in LIR. So, a translator from LIR to C program is implemented to debug programs. This translator (called LirToC) is not part of the SSA form optimizer, but it is provided as part of the COINS platform.

LIR instructions are somewhat like assembler and many of its operators correspond to C operators. However, LIR lacks some information making it difficult to translate it back to the input source program. LirToC uses the following particular translation rules to make up for this.

- conditional branch instruction
- array
- variable declaration

A detailed description follows.

8.1 Conditional Branch Instruction

Conditional branch instructions in LIR include JUMPC and JUMPN. JUMPC is a two-way and JUMPN is a multi-way branch instruction. They correspond to "if statement" and "switch statement" in C, respectively [1].

[1] Although statements other than if statements can be translated to JUMPC, translating it back to if statements retains the original meaning of the program.

LirToC translates all JUMPC to if-else. Figure 8.1 and Figure 8.2 show examples of translations of "JUMPC" and "JUMPN", respectively.

```
(LIR representation)

    (JUMPC (TSTLTS I32 (REG I32 "i.3%_1") (REG I32 "n.1%_1"))
           (LABEL I32 ".L4")
           (LABEL I32 ".L15"))

(C program representation after translation)

    if (i_3__1 < n_1__1) {
        goto _L4;
    }
    else {
        goto _L15;
    }
```

Figure 8.1: Translation example of conditional JUMP (JUMPC)

```
(LIR representation)

    (JUMPN (REG I32 "a.1%_3")
           (((INTCONST I32 1) (LABEL I32 ".L6"))
            ((INTCONST I32 2) (LABEL I32 ".L7"))
            ((INTCONST I32 3) (LABEL I32 ".L8"))
            ((INTCONST I32 4) (LABEL I32 ".L9")))
           (LABEL I32 ".L10"))

(C program representation after translation)

    switch(a_1__3) {
        case  1: goto _L6;break;
        case  2: goto _L7;break;
        case  3: goto _L8;break;
        case  4: goto _L9;break;
        default: goto _L10;
    }
```

Figure 8.2: Translation example of conditional JUMP (JUMPN)

8.2 Array

LIR does not have the concept of array. LIR uses MEM expressions to express memory references in input language. Thus

$key = a[j];$

and

$key = *(a + j);$

in the C language are represented by the same LIR instruction if a is declared as int. So, we have decided that all items represented by MEM expressions in LIR are translated to forms using pointers in C. Since address calculations in LIR are all made in 8-bit (1 byte) units, they are all translated to do the cast (unsigned char *) before calculation in C. Figure 8.2 shows a translation example.

```
(Representation in C)

    key = a[j]; or key = *(a+j);

(LIR representation)

    (SET I32 (REG I32 "key.4%_1")
             (MEM I32 (ADD I32 (FRAME I32 "A.1")
                               (LSHS I32 (REG I32 "j.2%_1")(INTCONST I32 2)))))

(Representation in C program after translation)

    key_4__1 = *(((unsigned char *)&(A_1) + (j_2__1 <<  2)));
```

Note: The array `A_1` is declared as a one-dimensional array

Figure 8.3: Example of translation of array

8.3 Declaration of Variables

As for the declaration of variables, the declaration is created using information in the local symbol table in each subprogram and the global symbol table.

Obtaining C types from the Ltype which is the type in LIR inherently depends on the compiling environment. However, currently they are processed using a fixed correspondence table as shown in Figure 8.1.

Table 8.1: Relation between Ltype and C language type

Ltype	C language type
I8	char
I16	short
I32	int
I64	long
F32	float
F64	double
F128	long double

Ltypes of arrays, structures and union take the form A+size such as A320. The size here is in bit units. To obtain the declaration of array and others from this information, the length of the array is calculated based on the element type (align) in the symbol table (expression 8.1).

$$(\text{array length}) = \frac{(\text{size})}{8 \times (\text{align})} \tag{8.1}$$

8.4 Known problems

LirToC has the following limitations.

- Function pointer
  Decoding of function pointer declarations has a problem and it cannot correctly translate into the C program.

- Pointer
  Pointer variables are handled as variables with L type I32 in the LIR symbol table. The symbol table does not contain any information as to whether it is a pointer variable, resulting in the declaration of an int variable. If any pointer is used in the original program, we must rewrite to provide declarations of pointer variables in the C program that LirToC generates.

- Signed / Unsigned
  The LIR symbol table does not distinguish between signed and unsigned variables. Thus the translated C program cannot express the difference between signed and unsigned variables. Furthermore, some LIR operations have two types signed / unsigned [2]. However, LirToC cannot express this difference.

- Initialization of global variables
  The current implementation of LirToC handles initialization and declaration of global variables only for character strings.

- ϕ-functions
  ϕ-functions cannot be expressed in a C program. Although formal descriptions of ϕ-functions can be used for translation, they are currently not translated.

[2] For example, DIVS and DIVU.

Appendix A

SSA Options

This appendix is included for user's convenience. The newest version is at 'doc-en/README.SSA.en.txt' of the distribution.

There are several compile time options for the SSA pass. For any other options of the COINS Compiler Driver, see 'READMEcc.txt' or 'doc-en/README.driver.en.txt'.

A.1 -coins:ssa-opt=xxx/yyy/.../zzz

Use SSA pass. This is necessary for using the SSA module. There are several optimizations in this module. To invoke the optimization, you should specify the optimizers with this option. Specified optimizers are invoked from left to right. First, as 'xxx' you MUST specify to which kind of SSA form you like to translate, such as minimal, semi-pruned or pruned. And then, as 'yyy' you can specify the optimizers which the SSA module invokes. You can specify the same optimizer two times, three times, and so on. Only optimizations that you specify are performed in that order. Finally, as 'zzz' you MUST specify how to back translate from SSA form.

The options are defined as follows:

Translation from normal form LIR to SSA form LIR :
(You MUST specify one of them at the beginning of this SSA option)

mini : Translation to Minimal SSA form

semi : Translation to Semi-Pruned SSA form

prun : Translation to Pruned SSA form (recommended for optimization)

Back Translation from SSA form LIR to normal form LIR :
(You MUST specify one of them at the end of this SSA option)

brig : Back translation using Briggs's Method

srd1 : Back translation using Sreedhar's Method I

srd2 : Back translation using Sreedhar's Method II

srd3 : Back translation using Sreedhar's Method III (recommentded for optimization)

(Options for coalescing are explained later)

Optimization :

cpyp : Copy Propagation

cstp : Constant Folding and Propagation with Conditional Branches

dce : Dead Code Elimination

cse : Common Subexpression Elimination

preqp : Global Value Numbering and Partial Redundancy Elimination with Efficient Question Propagation

hli : Hoisting Loop-invariant Code

osr : Operator Strength Reduction related to Induction Variables and Linear Function Test Replacement

ssag : Making SSA graph

divex : Divide Expression to Three-Address Code (the right-hand side of assignment will have only one operator)

gra : Global Reassociation

ebe : Empty Block Elimination

rpe : Redundant ϕ-function Elimination

cbb : Concatenate Basic Blocks

esplt : Split Critical Edge

lir2c : Make C program from LIR

Example :

If you specify the option

```
-coins:ssa-opt=prun/cstp/cse/srd3
```

the SSA module performs the following in that order:

1. make puruned SSA form
2. invoke constant folding and propagation with conditional branches
3. invoke common subexpression elimination
4. back translate using Sreedhar's Method III

A.2 -coins:ssa-no-change-loop

Before the optimizations, the SSA module changes the structure of the loops as follows, by default. This is for making effective loop optimization.

1. merge the several loops that have the same header block
2. insert the preheader
3. change the loop structure from 'while' type to 'do-while' type (precisely 'if-do-while'). The 'while' type is a loop such that the header and exit block of the loop are the same block.

The above is performed by default. If you DO NOT want to do that, specify this option.

A.3 -coins:ssa-no-copy-folding

During the translation to the SSA form, the SSA module removes and propagates the copy assignment statements such as 'x←y', by default.

If you DO NOT want to do that, specify this option.

A.4 -coins:ssa-no-redundant-phi-elimination

The SSA module eliminates redundant ϕ-functions after the translation to the SSA form, by default. A ϕ-function is redundant if:

1. the names of the target and the arguments of the ϕ-function are all the same as follows:

   $$x1 \leftarrow \phi(x1, x1, x1)$$

2. the names of the arguments of the ϕ-function are all the same as follows:

   $$x1 \leftarrow \phi(y1, y1, y1)$$

3. there are also other cases as follows:

   $$x1 \leftarrow \phi(y1, y1, x1) \quad \text{or} \quad x1 \leftarrow \phi(y1, y1, \bot)$$

In the first case, the SSA module just eliminates the ϕ-function. In the second case, the SSA module eliminates the ϕ-function and replaces uses of 'x1' by 'y1' in the statements which are evaluated after the ϕ-function. For further details, see [reference 1].

If you DO NOT want to do that, specify this option.

A.5 -coins:ssa-no-sreedhar-coalescing

During the back translation from SSA form by Sreedhar's method, the SSA module coalesces copy-related variables in SSA form, by default. This coalescing is proposed by Sreedhar and is called the SSA-based coalescing. This coalescing module is usually unified with Sreedhar's algorithm for back translation from SSA form. But for researchers' convenience, the SSA module can avoid it.

If you DO NOT want to do SSA based coalescing, specify this option.

A.6 -coins:ssa-with-chaitin-coalescing

Perform coalescing proposed by Chaitin after the back translation to normal form. This coalesces copy-related variables whose live ranges do not interfere each other. In general, after the back translation from SSA form, there may be some copy assignment statements in the program. Some copy assignment statements only change the names of variables, that is, they are useless. Coalescing these variables eliminates the useless copy assignment statements. This optimization is done in normal form LIR after the back translation from SSA form.

If you WANT to do that, specify this option. (This coalescing can be specified after any back translation method. But it may have no effect after the back translation by Sreedhar's Method III since that method does not insert copy assignment statements which can be coalesced by Chaitin's coalescing.)

A.7 -coins:ssa-no-memory-analysis

When Common Subexpression Elimination and/or Global Value Numbering and Partial Redundancy Elimination with Efficient Question Propagation are specified, the SSA module makes a simple alias analysis of memory accesses, by treating the whole memory as a single entity. (cf. section 8 of [reference 1])

If you DO NOT want to do that, specify this option.

A.8 -coins:ssa-no-replace-by-exp

Just before the back translation from SSA form, the SSA module finds the local variables, which are not "live out" from the current basic block and are used only once in the current basic block, and replaces the variables by the expressions which define the variables. (cf. section 5.4.6 "preprocessing for temporary variables" of [reference 1])

If you DO NOT want to do that, specify this option.

A.9 -coins:trace=SSA.xxxx

To output the trace information of this SSA module for debugging, specify the trace level as follows:

1 : Output only the message that the SSA module is invoked

100 : Output the agenda of the SSA module

1500 : Output two kinds of information:

(a) The inserted ϕ-functions when the SSA module translates normal LIR into SSA form.

(b) The inserted copy assignment statements when the SSA module back translates SSA form into normal LIR.

2000 : Output general debug information of all optimizers in the SSA module

10000 : Output much information about Sreedhar's Method III

The trace information includes the levels less than or equal to what you specified. If you specify

```
trace=SSA.1500
```

then the SSA module outputs information related to the level '1', '100' and '1500'.

A.10 -coins:ssa-opt=.../dump/...

For compiler developers, the SSA module provides the option 'dump' for debugging. This option should be specified within 'ssa-opt'. When this option is specified, the SSA module outputs the current LIR into the standard output.

For example, if the option is specified as follows:

```
-coins:ssa-opt=prun/dump/srd3/dump
```

the SSA module outputs the LIR

(1) after translation into the pruned SSA form, and

(2) after back translation from SSA form.

A.11 References

[1] 'Optimization in Static Single Assignment Form - External Specification' which is available on the web page 'http://www.coins-project.org/'.

Bibliography

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

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java second edition*. Cambridge University Press, 2002.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[5] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, July 1998.

[6] Preston Briggs, Tim Harvey, and Taylor Simpson. Static single assignment construction, version 1.0, January 1996. ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps.

[7] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Notices 17(6), Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction*, pages 98–105, 1982.

[8] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *Proceedings of the 1996 International Conference on Compiler Construction*, pages 223–237, April 1996.

[9] COINS project. http://www.coins-project.org/.

[10] Keith D. Cooper, L. Taylor Simpson, and Christpher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, September 2001.

[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. *ACM Transactions on Programming Languages and Systems*, 13(4):461–486, October 1991.

[12] Masaki Kohama. Comparison and evaluation of SSA normalization algorithms. *Master's Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, (in Japanese)*, 2004.

[13] Ikuo Nakata. *Structure and Optimization of Compilers (in Japanese).* Asakura-shoten, 1999.

[14] Berry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.

[15] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, January 1995.

[16] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. *SAS'99 LNCS 1694*, pages 194–210, 1999.

[17] Munehiro Takimoto and Kenichi Harada. Efficient question propagation.

[18] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(4):181–210, April 1991.