Translating Out of Static Single Assignment Form

Pushpinder Kaur Chouhan

January 8, 2003

1 Introduction

Static Single Assignment(SSA) form is an intermediate representation that compilers use to facilitate program analysis and optimization. Developed by researchers at IBM in the 80's. Each variable is only defined once, a redefinition of a variable is considered to be a new variable. The one (static) definition-site may be in a loop that is executed many(dynamic) times, thus the name static single assignment form. SSA represents the data flow and control flow properties of programs.

$$\begin{tikzcd}
\text{Original Intermediate Code} \arrow[d] & & & & \text{Optimized Intermediate Code} \\
P_0 \arrow[r] & P_1 \arrow[r] & P_2 \arrow[r] & \cdots\cdots\cdots & P_n \arrow[u]
\end{tikzcd}$$

Figure 1. Vertical arrows represent translation to/from SSA form. Horizontal arrows represent optimizations.

The role of SSA form in a compiler is illustrated in Figure 1. The intermediate code is put into SSA form, optimized in various ways and then translated back out of SSA form.

In the paper a new framework for leaving SSA form and a notion of a phi congruence class to facilitate the removal of phi instructions has been introduced. Three methods have been presented, of varying sophistication for placing copies. First method is closely related to Cytron et al. proposed simple algorithm for removing a k-input phi instruction except that it correctly eliminates copies even when transformations such as copy folding and code motion have been performed on the SSA form of a program. Second method is better then first method as it uses interference graph to guide copy instruction thus in turn reduces the number of many more copies in first method. Third method is best method among all as it uses both liveness and interference information to correctly eliminate phi instructions.

A new CSSA-based coalescing algorithm is also presented. The algorithm can eliminate redundant copies even when the source resource (or variable) and the destination resource interfere with each other when certain constraints are satisfied. This algorithm also uses phi congruence classes to eliminate redundant copies.

Experimental results show that third method is most effective in terms of number of copies in final code. Third method is faster than first method and places 89 percent fewer copies as compared to the first method.New CSSA-based algorithm also uses phi congruence classes to correctly and aggressively eliminate copies.

2 Keywords

To understand the methods presented in the paper we should understand the following keywords:

2.1 Phi Congruence Class

For a resource x let *phiConnectedResource*(x) $=\{y \mid$ x and y are referenced (ie used or defined) in the same phi instruction$\}$. Phi congruence class is defined as *Phi Congruence Class*$[x]$ to be the reflexive and transitive closure of phiConnectedResource(x). The phi congruence class of a resource represents a set of resources connected via phi instructions. *Phi Congruence Property* states that the occurrences of all resources which belong to the same phi congruence class in a program can be replaced by representative resource. After the replacement, the phi instruction can be eliminated without violating the semantics of the original program.

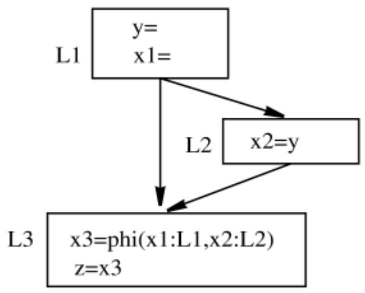

Figure 2. An example program in SSA form.

2.2 Liveness and Interference

A variable x is *live* at a program point p if there exists a path from p to a use of x that contains no definition of x. Two variables in a program are said to *interfere* if their live ranges overlap at any program point.

Each source operand in a phi instruction is a pair, x:L, where x is a resource name and L represents the control flow predecessor basic block label through which the value of x reaches the phi instruction. Let a phi instruction x0 = phi(x1:L1,x2:L2,. . . ,xn:Ln) is textually appear in a basic block L0, each use of xi where 1 $\leq$ i $\leq$ n . In the paper phi instruction is associated at the beginning of the basic block where the phi instruction textually appears, hence x0 is treated live upon entering L0. Following notations are used to describe the liveness properties at the beginning and at the end of each basic block, respectively.

- LiveIn[L]: The set of resources that are live at the beginning of the basic block L.
- LiveOut[L]: The set of resources that are live at the end of the basic block L.

3 Out of SSA Form

Once we have finished the translation to SSA form, we need to eliminate the phi functions, since they are only a conceptual tool and are not computationally effective. As no computer has a hardware that has phi instruction thus the compiler must translate the semantics of the phi instruction into commonly implemented instructions.

```
while ( . . . ) do


   read x
   y ← x+y
   x ← 6
   y ← x+y
end
```

(a)

```
while(. . .)do
   $y_3 \leftarrow \Phi(y_0, y_2)$
   $x_3 \leftarrow \Phi(x_0, x_2)$
   read $x_1$
   $y_1 \leftarrow x_1 + y_3$
   $x_2 \leftarrow 6$
   $y_2 \leftarrow x_2 + y_1$
end
```

(b)

```
$x_2 \leftarrow 6$
while(. . .)do
   $y_3 \leftarrow \Phi(y_0, y_2)$
   $x_3 \leftarrow \Phi(x_0, x_2)$
   read $x_1$
   $y_1 \leftarrow x_1 + y_3$

   $y_2 \leftarrow x_2 + y_1$
end
```

(c)

Figure 3. Program that really uses two instances for a variable after code motion.
(a) Source program; (b) unoptimized SSA form; (c) result of code motion

At first, it might seem possible to simply map all occurrences of xi back to x and to delete all of the phi instructions. However, the new variables introduced by translation to SSA form cannot always be eliminated, because optimizations may have capitalized on the storage independence of the new variables. The useful persistence of the new variables introduced by translation to SSA form can be illustrated by code motion in Figure 3. The source code (Figure 3a) assigns to x twice and uses it twice. The SSA form (Figure 3b) can be optimized by moving the invariant assignment out of the loop, yielding a program with separate variables for separate purposes (Figure 3c).

The dead assignment to x3 will be eliminated. These optimizations leave a region in the program where x1 and x2 are simultaneously live. Thus, both the variables are required. The original variable x cannot substitute for both renamed variables.

Algorithm for translating out of the SSA form consists of three steps:
1: Translating the TSSA form to a CSSA form
2: Eliminating redundant copies
3: Eliminating phi instructions and leaving the CSSA form.

3.1 Translating the TSSA form to a CSSA form

TSSA form and CSSA form are defined in the article to explain the methods presented below. The SSA form transformed to a state in which there are phi resource interference is called transformed SSA (TSSA) form and the SSA form that has the phi congruence property is called Conventional SSA (CSSA) form. The process of translating TSSA form to CSSA form ensures that none of the resources referenced within a phi instruction interfere with each other. Three methods are presented below for translating a TSSA form to CSSA form.

3.1.1 Method I - Naive Translation

In this method we insert the copies for all resources referenced in a phi instruction. One copy is inserted for each source resource in the corresponding basic block feeding the value to the phi instruction, and one copy is inserted in the same basic block as the phi instruction for the target resource. The redundant copies can be eliminated using CSSA-based coalescing algorithm described in later section(3.2).

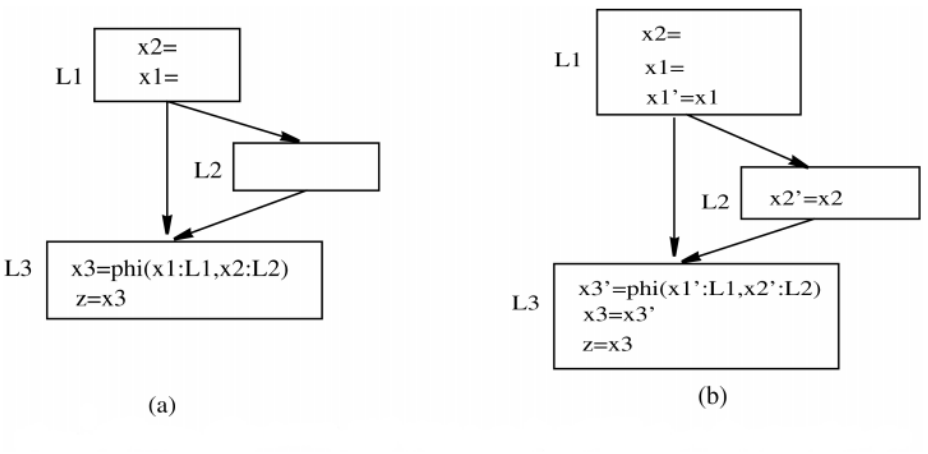

Figure 4. An example of translating TSSA form(a) to CSSA form(b) using method I.

Good feature of method I- This method is very simple to implement. It is as simple as Cytron et al. simple algorithm and it correctly eliminates copies even when transformations such as copy folding and code motion have been performed on the SSA form of a program.
Drawbacks- This method introduces many redundant copies. This method does not use either liveness or interference information to guide insertion and placement and thus places many more copies then necessary.

3.1.2 Method II - Translation Based on Interference Graph

In this method copies are inserted only if resources of phi instruction interfere. Considering example shown in Figure 4(a), we can see that x1 and x2 interfere with each other. To eliminate the interference we insert two copies, one for x1 and one for x2. Since x3 does not interfere with either x1 or x2 no new copy is inserted for x3. Then we incrementally update the interference graph and the phi congruence classes.

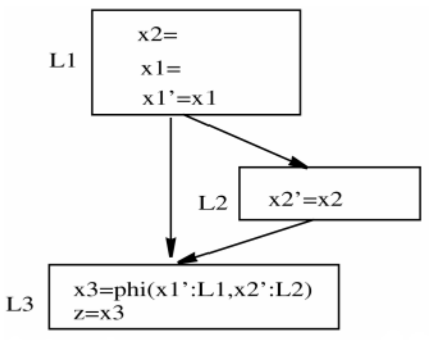

Figure 5. An example of translating TSSA form 4(a) to CSSA form using method II.

Good feature of method II- This method uses interference graph to guide copy instruction thus in turn reduces the number of copies to correctly eliminate the phi instruction. It is better than first method.
Drawback- This method still places more copies then necessary as it does not uses interference information.

3.1.3 Method III- Translation Based on Data Flow and Interference Graph Updates

An interference graph does not carry control flow information, so we may insert more copies than necessary. Thus liveness information should be used in addition to the interference graph to further reduce the number of copies that are necessary to eliminate the phi resource interference. We use LiveOut sets to eliminate interference among phi source resources. To eliminate interferences between the target resource and a source resource in a phi instruction we use LiveIn and LiveOut sets. Considering the same example shown in Figure 4(a). LiveOut[L1]= $\{x1, x2\}$ and LiveOut[L2]=$\{x2\}$. We can observe that x2 is in both LiveOut [L1] and [L2] and so we have to insert a new copy in both L1(x1'=x1) and L2(x2'=x2) because target of new copy (x1' = x1) in L1 will interfere with x2. Now since x1 is not in LiveOut[L2], we can eliminate the phi interference by inserting a new copy x2'=x2 only in L2 and no new copy(x1'=x1) is needed in L1.

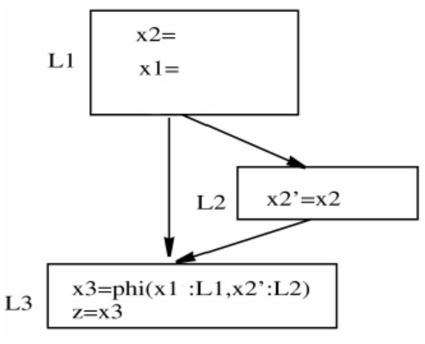

Figure 6. An example of translating TSSA form 4a to CSSA form using method III.

Good feature of method III- Any copies that this method places cannot be eliminated by the standard interference graph based coalescing algorithm.The source resources and target resources of the copies inserted by this method will interfere when we leave the SSA form.
Drawback- This method does not ensure that the number of copies inserted to eliminate phi resource interference is minimum.

- Complete Algorithm for Method III

Algorithm A: Algorithm for eliminating phi resource interferences based on data flow and interference graph updates.
eliminatePhiResourceInterference()
Inputs: instruction stream, control flow graph (CFG), LiveIn and LiveOut sets, interference graph.
Outputs: instruction stream, LiveIn and LiveOut sets, interference graph, phi congruence classes.

```
{
1: for each resource, x, participated in a phi
       phiCongruenceClass[x] = {x};
2: for each phi instruction (phiInst) in CFG {
       phiInst in the form of x0 = f(x1:L1, x2:L2, ..., xn:Ln);
       L0 is the basic block containing phiInst;
3: Set candidateResourceSet;
   for each xi, 0<=i<=n, in phiInst
       unresolvedNeighborMap[xi] = {};
4: for each pair of resources xi:Li and xj:Lj in phiInst, where
```

```
        0<= i, j <= n and xi != xj, such that there exists yi in
        phiCongruenceClass[xi], yj in phiCongruenceClass[xj],
        and yi and yj interfere with each other, {
        Determine what copies needed to break the interference
        between xi and xj using the four cases described in
        Section 3.1.3.
    }
5: Process the unresolved resources (Case 4) as described in
   Section 3.1.3.
6: for each xi in candidateResourceSet
        insertCopy(xi, phiInst);
7: // Merge phiCongruenceClass's for all resources in phiInst.
   currentphiCongruenceClass = {};
   for each resource xi in phiInst, where 0 <= i <= n {
        currentphiCongruenceClass += phiCongruenceClass[xi];
        Let phiCongruenceClass[xi] simply point to
        currentphiCongruenceClass;
        }
   }
8: Nullify phi congruence classes that contain only singleton
   resources.
}

insertCopy(xi, phiInst)
{ if( xi is a source resource of phiInst)
  {
   for every Lk associated with xi in the source list of phiInst
    {
    Insert a copy inst: xnew_i = xi at the end of Lk;
    Replace xi with xnew_i in phiInst;
    Add xnew_i in phiCongruenceClass[xnew_i]
    LiveOut[Lk] += xnew_i;
    if(for Lj an immediate successor of Lk, xi not in LiveIn[Lj]
     and not used in a phi instruction associated with Lk in Lj)
        LiveOut[Lk] -= xi;
    Build interference edges between xnew_i and LiveOut[Lk];
    }
   }else
   {
    // xi is the phi target, x0.
    Insert a copy inst: x0 = xnew_0 at the beginning of L0;
    Replace x0 with xnew_0 as the target in phiInst;
    Add xnew_0 in phiCongruenceClass[xnew_0]
    LiveIn[L0] -= x0;
    LiveIn[L0] += xnew_0;
    Build interference edges between xnew_0 and LiveIn[L0];
    }
}
```

- Explanation of the algorithm

The first step in algorithm is to initialize the phi congruence classes such that each resource in the phi instruction belongs to its own congruence class. These classes will be merged after eliminating interferences among them. The main feature of algorithm is to first check whether for any pair of resources, xi:Li and xj:Lj in a phi instruction, where $0 \leq$ i,j $\leq$ n, n is the number of phi resources operands, and xi $\neq$ xj, there exists resource yi in phiCongruenceClass[xi], yj in phiCongruenceClass[xj] and yi and yj interfere. If so we will insert copies to ensure that xi and xj will not be put in the same phi congruence class. Consider the case in which both xi and xj are source resources in the phi instruction. There are four cases to consider to insert copies instructions for resources in the phi instruction.
CASE 1. The intersection of phiCongruenceClass[xi] and LiveOut[Lj] is not empty, and the intersection of phiCongruenceClass[xj] and LiveOut[Li] is empty. A new copy, xi'=xi, is needed in Li to ensure that xi and xj are put in different phi congruence classes. So xi is added to candidateResourceSet.
CASE 2. The intersection of phiCongruenceClass[xi] and LiveOut[Lj] is empty, and the intersection of phiCongruenceClass[xj] and LiveOut[Li] is not empty. A new copy, xj'=xj, is needed in Lj to ensure that xi and xj are put in different phi congruence classes. So xj is added to candidateResourceSet.
CASE 3. The intersection of phiCongruenceClass[xi] and LiveOut[Lj] is not empty, and the intersection of phiCongruenceClass[xj] and LiveOut[Li] is not empty. Two new copies, xi'=xi in Li and xj'=xj in Lj, are needed to ensure that xi and xj are put in different phi congruence classes. So xi and xj are added to candidateResourceSet.
CASE 4. The intersection of phiCongruenceClass[xi] and LiveOut[Lj] is empty, and the intersection of phiCongruenceClass[xj] and LiveOut[Li] is empty. Either a copy, xi'=xi in Li, or a copy, xj'=xj in Lj, is sufficient to eliminate the interference between xi and xj. However, the final decision of which copy to insert is deferred until all pairs of interfering resources in the phi instruction are processed.

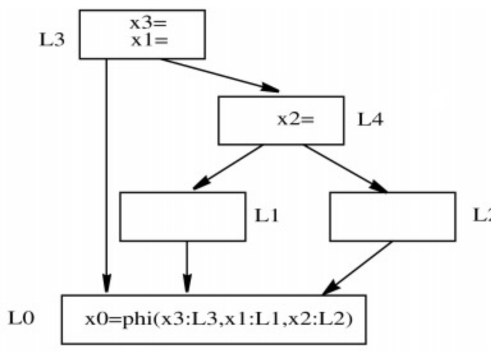

Figure 7. An example to illustrate the algorithm of Method III.

By deferring copy placement in CASE 4, placing redundant copies can be avoided. To see this consider the TSSA program shown in Figure 7. Initially phiCongruenceClass[x1] = $\{x1\}$, phiCongruenceClass[x2] = $\{x2\}$, and phiCongruenceClass[x3] = $\{x3\}$. The LiveOut sets for L1=$\{x1\}$, L2=$\{x2\}$, and L3=$\{x1, x3\}$. Since the live ranges of x1 and x2 interfere new copies are needed to break the phi resource interference. We can see that x1 is not in LiveOut[L2] and x2 is not in LiveOut[L1]. Therefore, we can eliminate the phi resource interference between them by inserting a copy either in L1 or in L2 according to CASE 4. Now rather than deferring the copy insertion let us insert a new copy x2'=x2 in L2. Since x1 and x3 also interfere with each other, copies are needed to eliminate the interference on this pair of resources. Here we can see that x1 appears in LiveOut[L3], so we must insert a new copy x1'=x1 in L1 to eliminate the phi resource interference between them according to CASE 1. By inserting this copy in L1 we can immediately see that the copy x2'=x2 inserted earlier is redundant. Therefore, to avoid inserting redundant copies we defer copy insertion for CASE 4 and keep track of resources for which we have not resolved copy insertion in a map called the unresolvedNeighborMap. Each time the copy insertion is deferred (unresolved) for a pair of resources xi and xj, we add xi to the set unresolvedNeighborMap[xj] and xj to the set unresolvedNeighborMap[xi]. Once all the resources in the phi instruction are processed, then processing of the unresolved resources is done. Resources are picked from the map in a decreasing size of unresolved resource set. For each resource x that is picked up from the map, x is added to candidateResourceSet if x contains at least one unresolved neighbor. We also mark x to be resolved and add x to candidateResourceSet. Finally, when all the maps are processed, it is possible a resource x that was marked as resolved may now contain all its neighbors to be marked as resolved. If this is the case, remove x from candidateResourceSet. Once all resources that need copies are put in the candidateResourceSet, insert copies for these resources as in Method I or Method II. Also update the liveness information, the interference graph, and the phi congruence class.

Illustration of the application of above algorithm to two problems, the 'lost-copy' problem and the 'swap' problem.

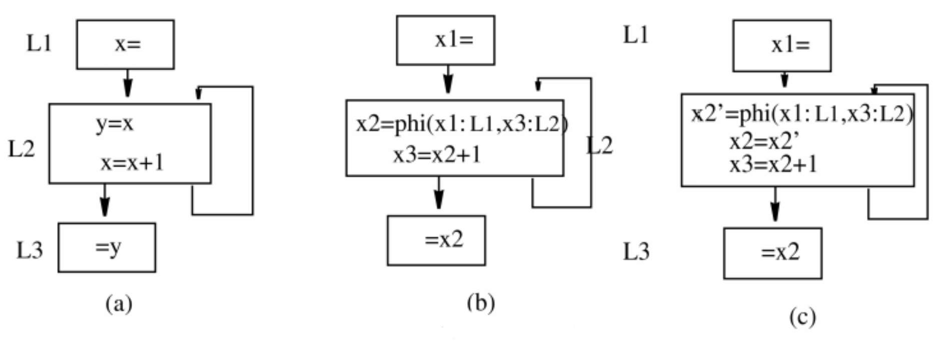

Figure 8. An example of of lost copy problem
(a) original code ; (b) TSSA form with copies folded; (c) CSSA form

Lost-Copy Problem: Lost copy problem is shown in Figure 8. If we use the algorithm in [2] to eliminate the phi instruction, the copy y=x would be lost in the translation. Let us apply above algorithm to this problem. From Figure 8(b) it is clear that x2 and x3 interfere, x2 is in LiveOut[L2], and x3 is not in LiveIn[L2]. So a new copy x2'=x2 is inserted in L2 for x2. This is shown in Figure 8(c). Now the phi instruction can be can simply eliminated after replacing references to all its resources by a representative resource.

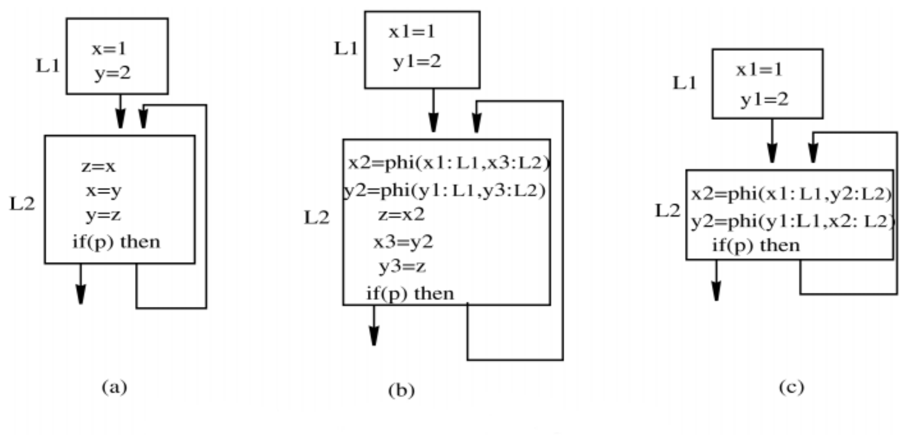

Figure 9. An example of the swap problem.
(a) Original code; (b) CSSA form (c) TSSA form with copies folded

Swap Problem: Swap problem is shown in Figure 9. Consider the TSSA form shown in Figure 9(c). First initialize the phi congruence classes of resources referenced in the two phi instructions by putting each resource in its own class. Next using liveness analysis, derive LiveOut[L2]=$\{x2, y2\}$ and LiveIn[L2]=$\{x2, y2\}$. Now consider the first phi instruction, where x2 and y2 interfere with each other. This result is shown in Figure 10(a).

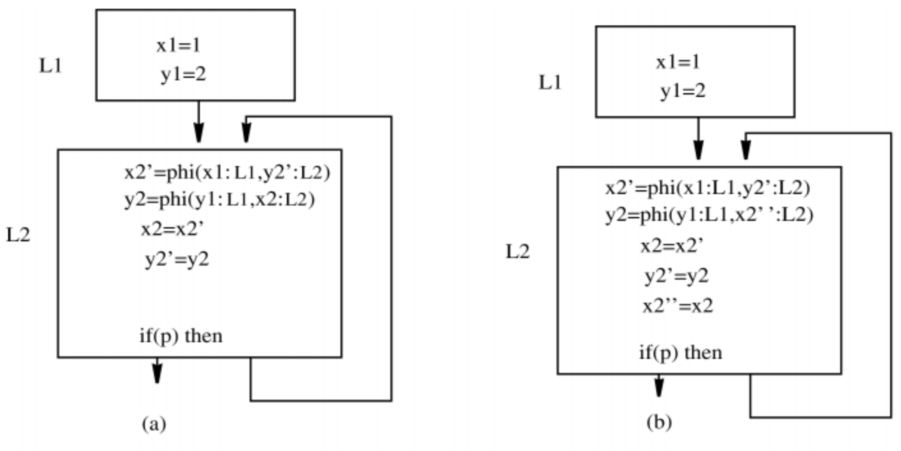

Figure 10. Breaking interference in the swap problem.

As y2 is in LiveIn[L2] and that x2 is in LiveOut[L2], so two copies has to be inserted: one for x2 and one for y2. After inserting the copies incrementally update the LiveIn set, the LiveOut set and the interference graph to reflect the new changes. The new LiveIn[L2]=$\{x2', y2\}$ and LiveOut[L2]= $\{x2, y2'\}$. And also update and merge the phi congruence classes for resources in the phi instruction so that resources x1, x2', and y2' are put in the same phi congruence class. Now consider the second phi instruction and notice that x2 and y2 still interfere. We can see that x2 is not in LiveIn[L2] and y2 is not in LiveOut[L2]. So only one copy is needed to eliminate the phi interference. This result is shown in Figure 10(b). Only three copies have been inserted and all three copies are essential.

3.2 Eliminating redundant copies

For eliminating the redundant copies **CSSA-based coalescing Algorithm** is introduced in the paper. If we us either algorithm given in [1]and [2] for eliminating the phi instructions, many redundant copies have been generated and to eliminate redundant copies another algorithm(Chaitin's coalescing algorithm) is used. We can use CSSA based coalescing algorithm instead of Chaitin's coalescing algorithm as we can eliminate a copy x=y even when their live ranges interfere, so long as the coalesced live range does not introduced any phi resource interference.
Let x=y be the copy that we wish to eliminate and assume that they are

not in the same phi congruence class. There are four cases to consider:
CASE 1: phiCongruenceClass[x]=={} and phiCongruenceClass[y]=={}. This means that x and y are not referenced in any phi instruction. The copy can be removed even if x and y interfere.
CASE 2: phiCongruenceClass[x]=={} and phiCongruenceClass[y]≠ {}. If x interferes with any resource in (phiCongruenceClass[y]-y) then the copy can not be removed, otherwise it can be removed.
CASE 3: phiCongruenceClass[x]≠ {}and phiCongruenceClass[y]== {}. If y interferes with any resource in (phiCongruenceClass[x]- x) then the copy can not be removed, otherwise it can be removed.
CASE 4: phiCongruenceClass[x]≠ {} and phiCongruenceClass[y]≠ {}. The copy cannot be removed if any resource in phiCongruenceClass[x] interferes with any resource in (phiCongruenceClass[y]-y) or if any resource in phiCongruenceClass[y] interferes with any resource in (phiCongruenceClass[x]-x), otherwise it can be removed.

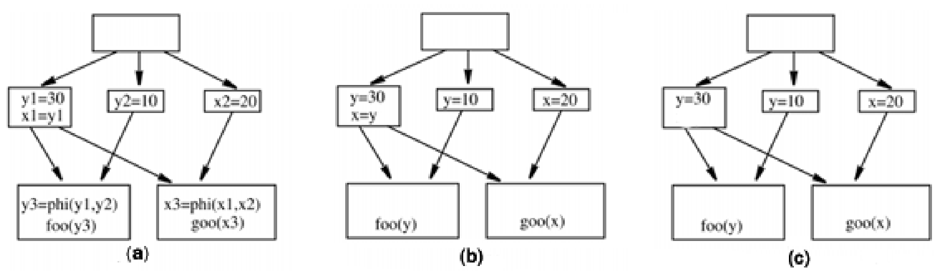

Figure 11. An example of copy elimination.
(a) CSSA form; (b) using Chaitin's algorithm; (c) using CSSA based coalescing algorithm

Now to explain CSSA-based coalescing algorithm, lets us consider one example shown in Figure 11. phiCongruenceClass $[x1] = \{x1, x2, x3\}$ and phiCongruenceClass $[y1] = \{y1, y2, y3\}$. We can see that x1 and y1 interfere, but we can still eliminate the copy according to CASE 4. After eliminating the copy, x1=y1, two phi congruence classes are merged. Since one of the resources in the phi congruence class interfere with each other we can eliminate the phi instruction by replacing all references to resources in the merged phi congruence class with a representative resource.

3.3 Eliminating phi instructions and leaving the CSSA form

After program transformations and optimization, a program in static single assignment form must be translated into some executable representation without phi instruction. It is easy to eliminate phi instruction from CSSA form. So for eliminating phi instruction from SSA, first convert the SSA form to a CCSA form using the method given in section(3.1).

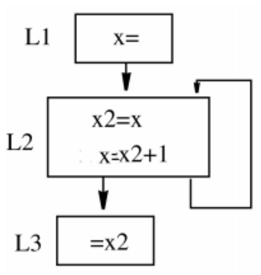

Figure 12. An example of translating out of SSA form.

From CSSA form simply eliminate the phi instruction by replacing all references in the phi instruction (belonging to the same phi congruence class) by a representative resource. To illustrate this, let us consider the example of lost copy problem shown in Figure 8(c). Resources x1,x2' and x3 that are referenced in the phi instruction belong to the same phi congruence class. Let x be the representative resource of the congruence class. Replace each reference to x1,x2' and x3 by x. After performing this transformation , phi instruction can be easily eliminated. This result is shown in Figure 12.

4 Experimental Results

Reducing the compilation time and space usage are the key motivations to develop the methods to translate out of SSA form. Thus for all three methods, we present two kinds of data: the first kind represents space usage and the second kind represents running time. The Number of copies inserted during the translation process is an indication of space usage. Experiments are done on a number of procedures taken from SPECint95 and one from Operating System source code.

Experiment shows that the translation Method II introduces 72.1 percent fewer copies then Method I and Method III introduces 89.9 percent fewer copies than method I. Thus both Method II and Method III outperform Method I in terms of space efficiency. By examining the total running time, method III performs significantly better than Method I in more than half of the cases and comparably in the rest. On an average Method III is about 15 percent faster than Method I. Although Method II has the fastest running time in some cases, this method is not very effective in reducing the number of copies in the final code.

After applying CSSA based coalescing to the CSSA form, for Method II after coalescing there are on an average 29.1 percent more copies than for method I and for Method III after coalescing there are 8.6 percent fewer copies than for Method I. In summary, conclusion is that Method II and Method III have better space efficiency than Method I and Method III is the most effective in terms of reducing the number of copies in final code. Method III is best among these three methods. **Note-** For details of experimental results you can see the original paper.

5 Conclusion

A uniform framework has been presented for eliminate phi instructions. Framework is based on two important properties: the phi congruence property and the property that none of the resources in a phi congruence class interfere. Liveness analysis and interference graph is used to eliminate the phi instructions. Framework does not use any structural properties of control flow graph or the dependence graph induced by the SSA graph to ensure that the copies are placed correctly. CSSA-based coalescing algorithm also uses phi congruence classes to correctly and aggressively eliminate the redundant copies.

In the paper all the presented methods are explained in detail and examples are also given to make understand each method. Comparison of the presented methods and algorithm is done with previously published methods (related to out of SSA form)and algorithm. Words and grammar used in the paper is very easy. There is no spelling mistake and no printing mistake in the paper. As the paper itself is very well explained, very few help reference is required to understand the concept given in it.

Having maximum good points, but then also the paper have something to comment on. Concepts are explained deeply but not systematically. Author explained all the steps of the algorithm(out of SSA form) but then also he wrote that he is not explaining the third step that he explain in the introduction of the paper. I think author was confused to give the name of the algorithm, as sometimes he wrote CSSA-based coalescing algorithm and next time SSA-based coalescing algorithm. Even third method of " translating TSSA form to CSSA form " is sufficient. When first and second methods are used CSSA-based coalescing algorithm has to be used to remove the redundant copies. And from experimental results also it is clear that third method is best, so it is better to directly use third method. Many definitions are redefined in the paper according to the author.

The paper is presented in detail and no new point is to be added, as it is complete. But improvements can be done in terms of reducing the number of copies inserted to correctly eliminate the phi instructions.

Referred Papers:

1. P. Briggs, K. Cooper, T. Harvey and Taylor Simpson. ”Practical Improvements to the Construction and Destruction of Static Single Assignment Form”.

2. R. Cytron, J.Ferrente, B.K. Rosen, M.N. Wegman and F.K. Zadeck. ”Effisiently Computing Static Single Assignment Form and the Control Dependence Graph”.

Referred Books:

1. Andrew W. Appel- Modern Compiler Implementation in ML

2. Steven S. Muchnick- Advanced Compiler Design Implementation

3. R. Allen & K. Kennedy- Optimizing Compilers for Modern Architectures

Referred Sites:

1. http://www.cs.ualberta.ca/∼ amaral/courses/680/webslides/T G-SSA/sld001.htm

2. http://www.informatik.uni-kiel.de/∼sacbase/faqs/compilerimpl ementationFAQ/node20.html

3. http://parasol-www.cs.tamu.edu/people/rwerger/Courses/605/ssa_ wt.ps

4. http://www-2.cs.cmu.edu/afs/cs/academic/class/15745-s02/www/lect ures/lect-ssa.pdf

5. http://www.cs.rice.edu/∼ keith/512/Lectures/08.pdf

6. http://www.flex-compiler.lcs.mit.edu/Harpoon/quads/node3.html

7. http://www.ececs.uc.edu/∼ ktomko/ACO/Lectures/Lecture_11 .pdf

8. http://www.cs.cornell.edu/courses/cs412/2002sp/lectures/lec36.ps