

Translating Out of Static Single Assignment Form

Vugranam C. Sreedhar¹, Roy Dz-Ching Ju², David M. Gillies³, and Vatsa Santhanam⁴

Performance Delivery Laboratory
Hewlett-Packard Company
11000 Wolfe Road
Cupertino, CA 95014, USA

Abstract. Programs represented in Static Single Assignment (SSA) form contain phi instructions (or functions) whose operational semantics are to merge values coming from distinct control flow paths. However, translating phi instructions into native instructions is nontrivial when transformations such as copy propagation and code motion have been performed. In this paper we present a new framework for translating out of SSA form. By appropriately placing copy instructions, we ensure that none of the resources in a phi congruence class interfere. Within our framework, we propose three methods for copy placement. The first method pessimistically places copies for all operands of phi instructions. The second method uses an interference graph to guide copy placement. The third method uses both data flow liveness sets and an interference graph to guide copy placement. We also present a new SSA-based coalescing method that can selectively remove redundant copy instructions with interfering operands. Our experimental results indicate that the third method results in 35% fewer copy instructions than the second method. Compared to the first method, the third method, on average, inserts 89.9% fewer copies during copy placement and runs 15% faster, which are significant reductions in compilation time and space.

1. Introduction

Static Single Assignment (SSA) form is an intermediate representation that facilitates the implementation of powerful program optimizations [7, 12, 13], where each program name is defined exactly once and phi (ϕ) instructions (or nodes) are inserted at confluent points to merge multiple values into a single name. Phi instructions are not directly supported on current architectures, and hence they must be eliminated prior to final code generation [7]. However, translating out of SSA form is nontrivial when certain transformations, such as copy folding and code motion, have been

¹ Vugranam C. Sreedhar is currently affiliated with IBM T. J. Watson Research Center, and his e-mail address is sreedhar@watson.ibm.com.

² The e-mail address of Roy D. C. Ju is royju@cup.hp.com.

³ David Gillies is currently affiliated with Programmer Productivity Research Center at Microsoft Research, and his e-mail address is dgillies@research.microsoft.com.

⁴ The e-mail address of Vatsa Santhanam is vatsa@cup.hp.com.

performed. Most of the previous work on SSA form have concentrated either on efficiently constructing the representation [7, 11], or on proposing new SSA-based optimization algorithms [4, 5, 12].

We are aware of the following published articles related to translating out of SSA form. Cytron et al. [7] proposed a simple algorithm for removing a k -input phi instruction by placing ordinary copy instructions (or assignments) at the end of every control flow predecessor of the basic block containing the phi instruction. Cytron et al. then used Chaitin's coalescing algorithm to reduce the number of copy instructions [3]. The work in [2] showed that Cytron et al.'s algorithm cannot be used to correctly eliminate phi instructions from an SSA representation that has undergone transformations such as copy folding and value numbering. To address this, Briggs et al. [2] proposed an alternative solution for correctly eliminating phi instructions. Briggs et al. exploit the structural properties of both the control flow graph and the SSA graph of a program to detect particular patterns, and use liveness information to guide copy insertions for eliminating phi instructions. Any redundant copies introduced during phi instruction elimination are then eliminated using Chaitin's coalescing algorithm [3]. Pineo and Soffa [10] used interference graph and graph coloring to translate programs out of SSA form for the purpose of symbolic debugging of parallelized code. Leung and George [8] constructed SSA form for programs represented as native machine instructions, including the use of machine dedicated registers. Upon translating out of SSA form, a large number of copy instructions, including many redundant ones, may be inserted to preserve program semantics, and they rely on a coalescing phase in register allocation to remove the redundant copy instructions.

In this paper we present a new framework for leaving SSA form and for eliminating redundant copies. We introduce the notion of a *phi congruence class* to facilitate the removal of phi instructions. Intuitively, a phi congruence class contains a set of resources (or variables) that will be given the same name when we translate out of SSA form. The key intuition behind our method for eliminating phi instructions is to ensure that none of the resources within a phi congruence class interfere among each other. The idea is very similar to coloring-based register allocation problem [3], where if two live ranges interfere they should be given two different physical registers. But if there is only one unused physical register available, then one of the live ranges should be spilled to eliminate the interference. To break the interferences among resources in a phi instruction we introduce "spill code" by placing copy instructions. Another unique aspect of our method is that we don't use any structural properties of either the control flow graph or the SSA graph to guide us in the placement of copy instructions.

We present three different methods of varying sophistication for placing copies. Our first method is closely related to the copy placement algorithm described in [7] 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. This method does not explicitly use either liveness or interference information to guide copy insertion and placement, and therefore places many more copies than necessary. To reduce the number of copies that are needed to correctly eliminate phi instruction, our second method uses an interference graph to guide copy placement. Although it places fewer copies than the first method, it still places more copies than necessary. To further reduce the number of copies, our third method uses both liveness and interference information to correctly eliminate phi instructions. A unique aspect of

this method is that any copies that it places cannot be eliminated by the standard interference graph based coalescing algorithm.

In this paper we also present a new SSA-based coalescing algorithm that 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.

We implemented all three methods for copy placement and also implemented our SSA based coalescing algorithm. Our experimental results indicate that our third method is most effective in terms of the number of copies in the final code, and it is also faster than the first method. Note that reducing the compilation time and space usage is a key motivation to develop a new method to translate out of SSA form. On average we found that our third method inserts 89.9% fewer copies compared to the first method. This also means that our third method is an improvement over the algorithms proposed by Cytron et al. [7] and Briggs et al. [2] (since they also pessimistically insert copies to eliminate phi instructions).

The rest of the paper is organized as follows. In the next section we will motivate the problem of translating out of SSA form, and introduce the notion of phi congruence class. Section 3 briefly discusses liveness analysis for programs under SSA form. Section 4 presents three different methods for placing copies. Section 5 presents a new SSA-based coalescing algorithm for eliminating redundant copies. Section 6 presents experimental results. Finally, Section 7 discusses related work and presents our conclusion.

2. Motivation and Phi Congruence Class

When the SSA form of a program is constructed using the algorithm of Cytron et al. [7], the representation has the following two properties:

- Resources (or variables) are appropriately renamed so that each resource has a single definition.
- Phi instructions are introduced to merge multiple definitions coming from distinct control flow paths. Each phi instruction has one input operand for each control flow predecessor.

Figure 1 shows a program in SSA form. 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 [6].

Given a resource x let $\text{phiConnectedResource}(x) = \{y \mid x \text{ and } y \text{ are referenced (i.e. used or defined) in the same phi instruction, or there exists a resource } z \text{ such that } y \text{ and } z \text{ are referenced in the same phi instruction and } x \text{ and } z \text{ are referenced in the same phi instruction}\}$. We define *phi congruence class* as $\text{phiCongruenceClass}[x]$ to be the reflexive and transitive closure of $\text{phiConnectedResource}(x)$. Intuitively, the phi congruence class of a resource represents a set of resources “connected” via phi instructions. The SSA form that is constructed using Cytron et al.’s algorithm (which we call the *Conventional SSA (CSSA) form*) has the following important property.

- **Phi Congruence Property.** The occurrences of all resources which belong to the same phi congruence class in a program can be replaced by a representative

resource. After the replacement, the phi instruction can be eliminated without violating the semantics of the original program⁵.

For the example shown in Figure 1, 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. We can replace each reference to x1, x2 and x3 by x. Once we perform this transformation, we can eliminate the phi instruction. The resulting program is shown in Figure 2.

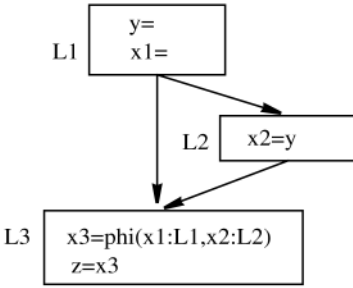


Figure 1. An example program in SSA form.

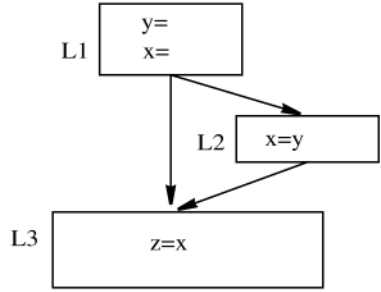


Figure 2. An example of translating out of CSSA form.

Referring back to the example shown in Figure 1, let us coalesce the copy $x2=y$. The resulting SSA form is shown in Figure 3. Now if we use the phi congruence property to replace the references of x1, x3, and y with a representative resource the resulting program will not preserve the semantics of the original program. This is because, after folding the copy $x2=y$, x1 and y have overlapping live ranges, and hence should not be replaced by the same name. Therefore any interfering resource in a phi congruence class should be “spilled” by inserting copies. In other words we should ensure that none of the resources in a phi congruence interfere with each other. The idea is very similar to coloring-based register allocation problem [3], where if two live ranges interfere they should be assigned different physical registers. But if there is only one unused physical register available, then one of the live ranges needs to be spilled to eliminate the interference.

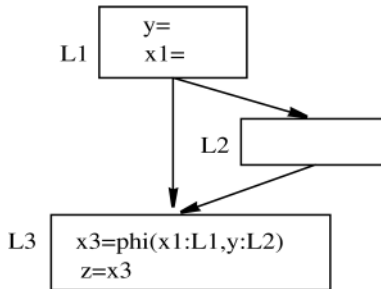


Figure 3. An example of TSSA form.

⁵ Phi congruence is analogous to the notion of register “webs” (outside of SSA form) defined in [9].

Given a CSSA form, optimizations such as copy folding and code motion, may transform the SSA form to a state in which there are phi resource interferences. Let us call such an SSA form *TSSA* (for *transformed SSA*) form. Our algorithm for translating out of the SSA form consists of three steps:

- Step 1: Translating the TSSA form to a CSSA form;
- Step 2: Eliminating redundant copies; and
- Step 3: Eliminating phi instructions and leaving the CSSA form.

Our approach⁶ to eliminating phi resource interferences in Step 1 relies on liveness analysis and interference detection, which is discussed in the next section. Step 2 is our new CSSA-based coalescing algorithm (see Section 5). Step 3 is a straightforward application of the phi congruence property and elimination of phi instructions (the details not presented).

3. Liveness and Interference

A variable v is *live* at a program point p if there exists a path from p to a use of v that contains no definition of v [1]. A traditional bit vector method can be used to analyze programs in SSA form for liveness by treating phi instructions specially. Cytron and Gershbein [6] made an important observation regarding phi instructions. Given a phi instruction $x_0 = \text{phi}(x_1:L_1, x_2:L_2, x_3:L_3, \dots, x_n:L_n)$ that textually appears in a basic block L_0 , each use of x_i , where $1 \leq i \leq n$, is associated with the end of the corresponding predecessor basic block through which x_i reaches L_0 . In this paper we associate the definition of a phi instruction to be at the beginning of the basic block where the phi instruction textually appears, i.e., basic block L_0 , and hence x_0 is treated live upon entering L_0 .

Given the above special treatment for phi instructions, we can use the traditional bit vector technique for liveness analysis, and construct the interference graph. Two variables in a program are said to interfere if their live ranges overlap at any program point. We will use the following notation to describe the liveness properties at the beginning and at the end of each basic block, respectively.

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

4. Translating TSSA Form to 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. Here we present three methods for translating a TSSA form to a CSSA form.

⁶ Although there are known SSA-based transformations that preserve CSSA form, e.g. [4], they often require the computations of additional data flow information, e.g availability and anticipatability. Furthermore, insisting on CSSA form may constrain some optimization opportunities.

4.1 Method I: Naive Translation

In this method we naively insert 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. Figure 4 illustrates this naive translation.

This method is very simple to implement but introduces many redundant copies. The redundant copies can be eliminated using our CSSA-based coalescing algorithm (see Section 5). It is important to note that when a resource of a phi instruction is spilled we will always ensure that the new resource is referenced only in the phi instruction. This spilling technique will also be used even for Method II and Method III translation.

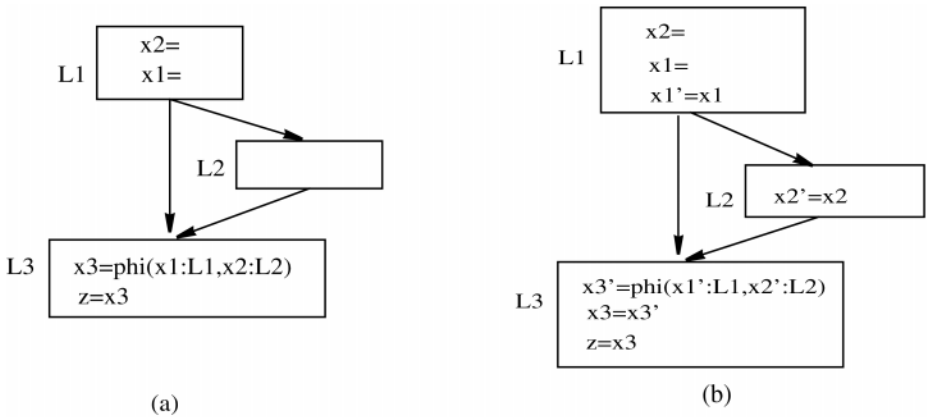


Figure 4. An example of translating TSSA form (a) to CSSA form (b) using Method I.

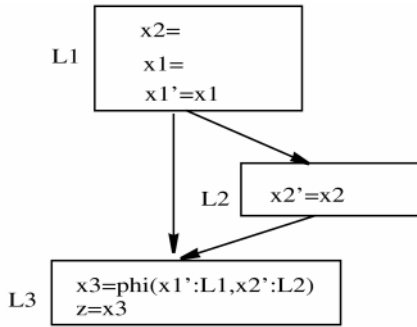


Figure 5. Translation to CSSA form using Method II.

4.2 Method II: Translation Based on Interference Graph Update

In this method we will insert copies only if resources of phi instructions interfere. Consider the example shown in Figure 4(a). Here we can see that $x1$ and $x2$ interfere

with each other. To eliminate the interference we insert two copies, one for x_1 and one for x_2 . (Since x_3 does not interfere with either x_1 or x_2 no new copy is inserted for x_3 .) Next we incrementally update the interference graph and the phi congruence classes. Applying this algorithm to the example in Figure 4(a) results in the CSSA form shown in Figure 5. Notice that we have inserted two copies. But it is evident from the figure that only the copy $x_2' = x_2$ is needed to ensure correctness. Since the interference graph does not carry control flow information, it is difficult to determine when copy instructions are redundant using the interference graph alone. Also, without the liveness information, the interference graph update will be conservative. In the next section we will show how to further reduce the number of copies that are needed to correctly eliminate phi instructions by using both interference and liveness information for guiding copy insertions.

It is important to observe that while deciding whether copy instructions are needed for x_1 and x_2 , in addition to checking the interference between x_1 and x_2 , we must also check for interferences with all other resources that will be replaced with the same names as x_1 and x_2 after translating out of SSA form (i.e., all members of their phi congruence class). We will elaborate on this in the next section.

4.3 Method III: Translation Based on Data Flow and Interference Graph Updates

Since an interference graph does not carry control flow information we may insert more copies than necessary in Method II to eliminate the phi resource interferences. In this section we will use liveness information in addition to the interference graph to further reduce the number of copies that are necessary to eliminate the phi resource interferences.

To motivate the new algorithm, consider Figure 4(a). Here x_1 and x_2 interfere, $\text{LiveOut}[L1] = \{x_1, x_2\}$, and $\text{LiveOut}[L2] = \{x_2\}$. Notice that x_2 is in $\text{LiveOut}[L1]$, and we claim that inserting a new copy $x_1' = x_1$ *only* in L1 will not eliminate the phi interference (i.e., we still need to insert a copy in L2 to eliminate the interference). This is because the target of the new copy will interfere with x_2 . Now since x_1 is not in $\text{LiveOut}[L2]$ we can eliminate the phi interference by inserting a new copy $x_2' = x_2$ *only* in L2 (i.e., a copy is not needed in L1). Notice that we used 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 (see the “lost-copy” problem and the “swap” problem discussed later in the section). The complete algorithm for eliminating phi resource interferences based on data flow and interference graph updates is given in the appendix. A unique aspect of this algorithm is that any copy it places cannot be eliminated by the standard interference graph based coalescing algorithm (assuming there are no dead phi instructions). In other words, the source and target resources of the copies inserted by this method will interfere when we leave the SSA form. Also, the algorithm precisely updates both the interference graph and liveness information. Note that this algorithm does not ensure that the number of copies inserted to eliminate phi resource interference is minimum. (We believe that the problem of ensuring a minimum number of copies inserted to correctly eliminate phi instructions is still to be formulated, e.g. based on a static count or a dynamic count, and remains unresolved yet.)

The first step in the 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 crux of the algorithm is to first check whether for any pair of resources, $x_i:L_i$ and $x_j:L_j$ in a phi instruction, where $0 \leq i, j \leq n$, n is the number of phi resources operands, and $x_i \neq x_j$, there exists resource y_i in $\text{phiCongruenceClass}[x_i]$, y_j in $\text{phiCongruenceClass}[x_j]$ and y_i and y_j interfere. If so we will insert copies to ensure that x_i and x_j will not be put in the same phi congruence class. Consider the case in which both x_i and x_j are source resources in the phi instruction.⁷ There are four cases to consider to insert copies instructions for resources in the phi instruction.

- Case 1. The intersection of $\text{phiCongruenceClass}[x_i]$ and $\text{LiveOut}[L_j]$ is not empty, and the intersection of $\text{phiCongruenceClass}[x_j]$ and $\text{LiveOut}[L_i]$ is empty. A new copy, $x_i' = x_i$, is needed in L_i to ensure that x_i and x_j are put in different phi congruence classes. So x_i is added to $\text{candidateResourceSet}$.
- Case 2. The intersection of $\text{phiCongruenceClass}[x_i]$ and $\text{LiveOut}[L_j]$ is empty, and the intersection of $\text{phiCongruenceClass}[x_j]$ and $\text{LiveOut}[L_i]$ is not empty. A new copy, $x_j' = x_j$, is needed in L_j to ensure that x_i and x_j are put in different phi congruence classes. So x_j is added to $\text{candidateResourceSet}$.
- Case 3. The intersection of $\text{phiCongruenceClass}[x_i]$ and $\text{LiveOut}[L_j]$ is not empty, and the intersection of $\text{phiCongruenceClass}[x_j]$ and $\text{LiveOut}[L_i]$ is not empty. Two new copies, $x_i' = x_i$ in L_i and $x_j' = x_j$ in L_j , are needed to ensure that x_i and x_j are put in different phi congruence classes. So x_i and x_j are added to $\text{candidateResourceSet}$.
- Case 4. The intersection of $\text{phiCongruenceClass}[x_i]$ and $\text{LiveOut}[L_j]$ is empty, and the intersection of $\text{phiCongruenceClass}[x_j]$ and $\text{LiveOut}[L_i]$ is empty. Either a copy, $x_i' = x_i$ in L_i , or a copy, $x_j' = x_j$ in L_j , is sufficient to eliminate the interference between x_i and x_j . However, the final decision of which copy to insert is deferred until all pairs of interfering resources in the phi instruction are processed.

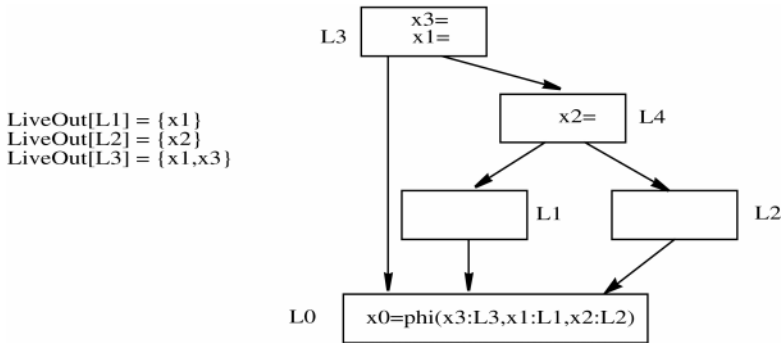


Figure 6. An example of TSSA form with LiveOut sets.

⁷ The situation is exactly the same when one of x_i or x_j is a target resource except that we use both LiveIn and LiveOut sets to decide copy placement.

By deferring copy placement in Case we avoid placing redundant copies. To see this consider the TSSA program shown in Figure 6. Initially $\text{phiCongruenceClass}[x1] = \{x1\}$, $\text{phiCongruenceClass}[x2] = \{x2\}$, and $\text{phiCongruenceClass}[x3] = \{x3\}$. The LiveOut sets for L1, L2, and L3 are shown in the figure. 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 $\text{LiveOut}[L2]$ and $x2$ is not in $\text{LiveOut}[L1]$. Therefore, we can eliminate the phi resource interference between them by inserting a copy either in L1 or in L2. 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 $\text{LiveOut}[L3]$, so we must insert a new copy $x1'=x1$ in L1 to eliminate the phi resource interference between them. 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 x_i and x_j , we add x_i to the set $\text{unresolvedNeighborMap}[x_j]$ and x_j to the set $\text{unresolvedNeighborMap}[x_i]$. Once all the resources in the phi instruction are processed, we handle the unresolved resources. We pick resources from the map in a decreasing size of unresolved resource set. For each resource x that is picked up from the map, we add x 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 we remove x from *candidateResourceSet*.

Once all resources that need copies are put in the *candidateResourceSet*, we insert copies for these resources as in Method I or Method II. We also update the liveness information, the interference graph, and the phi congruence class. The details of these updates are given in the appendix. Next we illustrate the application of our algorithm to two problems, the ‘lost-copy’ problem and the ‘swap’ problem, discussed in [2].

The Lost-Copy Problem. Figure 7 illustrates the lost copy problem. Figure 7(a) shows the original code. Figure 7(b) shows the TSSA form (with copies folded). If we use the algorithm in [7] to eliminate the phi instruction, the copy $y=x$ would be lost in the translation.

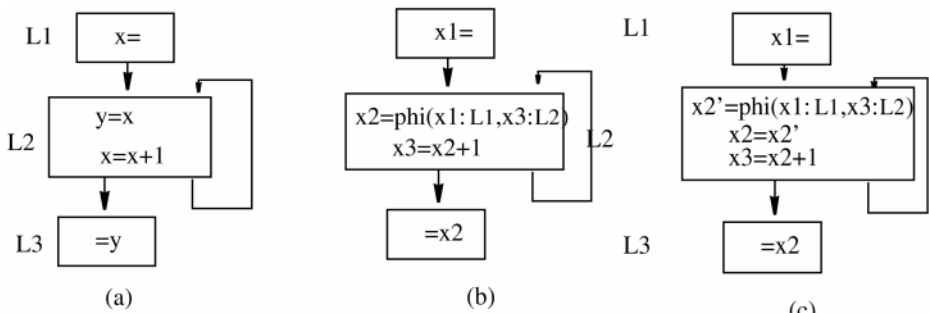


Figure 7. An example of the lost copy problem.

Let us apply our algorithm to this problem. From Figure 7(b) we can see that x_2 and x_3 interfere, x_2 is in $\text{LiveOut}[L_2]$, and x_3 is not in $\text{LiveIn}[L_2]$. So a new copy $x_2=x_2'$ is inserted in L_2 for x_2 . Once we do this as shown in Figure 7(c) we can simply eliminate the phi instruction after replacing references to all its resources by a representative resource.

The Swap Problem. Let us apply the algorithm to the swap problem in [2] shown in Figure 8. The original code for the swap problem is given in Figure 8(a), and the corresponding CSSA form is given in Figure 8(b). After we perform copy folding on the CSSA form, we get the TSSA form shown in Figure 8(c).

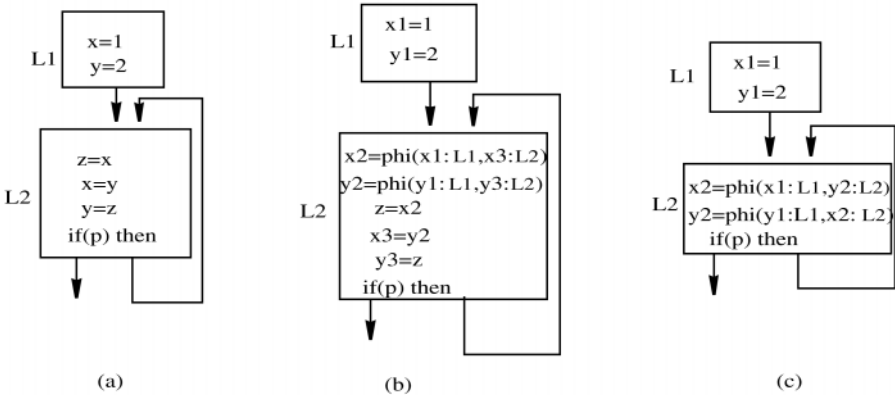


Figure 8. An example of the swap problem.

Consider the TSSA form shown in Figure 8(c). We first initialize the phi congruence classes of resources referenced in the two phi instructions by putting each resource in its own class (e.g., $\text{phiCongruenceClass}[x_1]=\{x_1\}$, $\text{phiCongruenceClass}[x_2]=\{x_2\}$, etc.) Next, using liveness analysis we can derive $\text{LiveOut}[L_2]=\{x_2, y_2\}$ and $\text{LiveIn}[L_2]=\{x_2, y_2\}$. Now consider the first phi instruction, where we can see that x_2 and y_2 interfere with each other. Note that the use of x_2 in the second phi instruction occurs at the end of L_2 (see Section 3 for details). Also, notice that y_2 is in $\text{LiveIn}[L_2]$ and that x_2 is in $\text{LiveOut}[L_2]$. Therefore we will insert two copies, one for x_2 and one for y_2 . The resulting program is shown in Figure 9(a). After inserting the copies we incrementally update the LiveIn set, the LiveOut set and the interference graph to reflect the new changes. The new $\text{LiveIn}[L_2]=\{x_2', y_2\}$ and $\text{LiveOut}[L_2]=\{x_2, y_2'\}$. We will also update and merge the phi congruence classes for resources in the phi instruction so that resources x_1 , x_2' , and y_2' are put in the same phi congruence class. Now consider the second phi instruction and notice that x_2 and y_2 still interfere. We can see that x_2 is not in $\text{LiveIn}[L_2]$ and y_2 is not in $\text{LiveOut}[L_2]$. So only one copy is needed to eliminate the phi interference. The resulting program is shown in Figure 9(b). We can see that we have inserted only three copies and all three copies are essential.

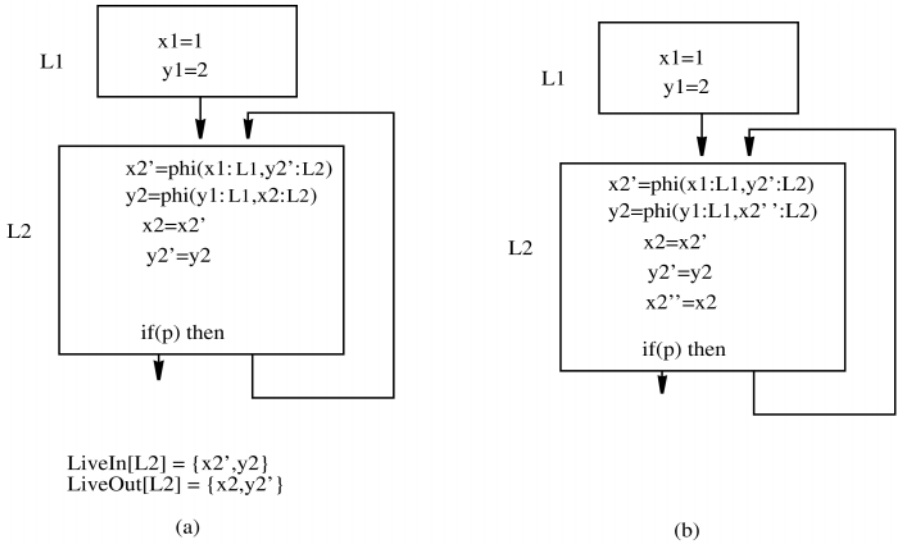


Figure 9. Breaking interferences in the swap problem.

5. SSA Based Coalescing

Once phi instructions have been eliminated the algorithms given in [7] and in [2] rely on Chaitin’s coalescing algorithm to eliminate as many redundant copies as possible. Consider the CSSA form shown in Figure 10(a). Since none of the resources within phi instructions interferes with each other, we can eliminate the phi instruction by using the phi congruence property. The resulting program is shown in Figure 10(b). Now let us apply Chaitin’s algorithm to eliminate the copy $x=y$ [3]. Since x and y interfere with each other this copy cannot be eliminated.

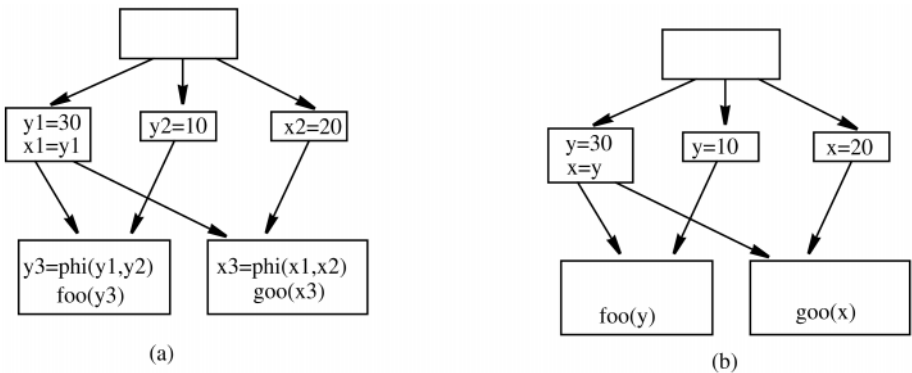


Figure 10. Copy elimination using Chaitin’s coalescing algorithm.

In this section we present a new CSSA-based coalescing algorithm that will allow us to eliminate the copy $x1=y1$. The key intuition behind our algorithm is that we can

eliminate a copy $x=y$ even when their live ranges interfere, so long as the coalesced live range does not introduce any phi resource interference.

Let $x=y$ be the copy that we wish to eliminate. Assume that they are not in the same phi congruence class. (It is trivial otherwise.) There are three cases to consider:

- Case 1: $\text{phiCongruenceClass}[x]=\{\}$ and $\text{phiCongruenceClass}[y]=\{\}$. 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: $\text{phiCongruenceClass}[x]=\{\}$ and $\text{phiCongruenceClass}[y]\neq\{\}$. If x interferes with any resource in $(\text{phiCongruenceClass}[y]-y)$ then the copy cannot be removed, otherwise it can be removed. The situation is similar but opposite for the case where $\text{phiCongruenceClass}[x]\neq\{\}$ and $\text{phiCongruenceClass}[y]=\{\}$.
- Case 3: $\text{phiCongruenceClass}[x]\neq\{\}$ and $\text{phiCongruenceClass}[y]\neq\{\}$. The copy cannot be removed if any resource in $\text{phiCongruenceClass}[x]$ interferes with any resource in $(\text{phiCongruenceClass}[y]-y)$ or if any resource in $\text{phiCongruenceClass}[y]$ interferes with any resource in $(\text{phiCongruenceClass}[x]-x)$, otherwise it can be removed.

Consider the example shown in Figure 10(a). From the figure we can see that $\text{phiCongruenceClass}[x1] = \{x1, x2, x3\}$ and $\text{phiCongruenceClass}[y1]=\{y1,y2,y3\}$. We can see that $x1$ and $y1$ interfere, but we can still eliminate the copy using Case 3. After eliminating the copy, $x1 = y1$, the two phi congruence classes are merged. Since none of resources in the phi congruence class interferes 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. The resulting program no longer has the copy, $x = y$. Our heuristic for handling Case 3 is conservative but safe. One can easily enhance this heuristic so that more copies can be eliminated while still ensuring that the phi congruence property is satisfied.

6. Experimental Results

To demonstrate the effectiveness of our approach we implemented all three methods for translating out of SSA form. We also implemented our CSSA based coalescing. The experimental results are summarized in Table 1. We ran our experiments on a number of procedures taken from SPECint95 and other application suites. Here we present our results for a set of ten representative procedures. Nine out of the ten procedures are from the SPECint95 suite, and one is from operating system source code. One typical characteristic of the ten procedures is that they are large. In our compiler, optimizations, such as global code motion and common sub-expression elimination, introduce phi resource interferences. All of the dead phi instructions have been pruned as part of our SSA construction phase.

Note that reducing the compilation time and space usage is a key motivation to develop the third method to translate out of SSA form. For all three methods we present two kinds of data; the first kind represents space usage and the second kind represents running time. In Table 1 BT indicates the number of copies prior to leaving the TSSA form, and AT indicates the number of copies after translating to the CSSA form. The difference between AT and BT gives the number of copies introduced during the translation of the TSSA form to the CSSA form. For the set of benchmarks

that we experimented with we found that during the translation Method II introduces 72.1% fewer copies than Method I, and Method III introduces 89.9% fewer copies than Method I. The number of copies inserted during the translation process is an indication of space usage. Thus, we can see that both Method II and Method III outperform Method I in terms of space efficiency.

AC in the table indicates the number of copies that exist after applying our CSSA based coalescing to the CSSA form. For both Method II and Method III we did not recompute the data flow information and the interference graph, but relied on the incremental update performed by the two methods. Since, for Method II, the interference graph update is conservative, after coalescing there are, on average, 29.1% more copies than for Method I. Interestingly, after coalescing there are 8.6% fewer copies using Method III compared to Method I. It is important to note that for Method III, the subsequent coalescing does not eliminate any copy instruction introduced during the TSSA-to-CSSA translation, but instead it eliminates only

Table 1: Empirical result for the three methods.

Procedure Name	BT	Meth ods	AT	AT-BT	AC	DF/IG (secs)	TT (secs)	TC (secs)	Total (secs)
Part_delete (vortex)	86	I	112	26	92	0.08	0.01	0.02	0.11
		II	100	14	99	0.09	0.01	0.01	0.11
		III	92	6	91	0.09	0.01	0.02	0.12
Yyparse (gcc)	723	I	918	195	140	2.63	0.04	0.50	3.17
		II	761	38	162	2.50	0.05	0.43	2.98
		III	738	15	139	2.64	0.03	0.42	3.09
Yylex (perl)	1060	I	2901	1841	590	3.51	0.28	2.05	5.84
		II	1309	249	652	2.77	0.60	0.78	4.15
		III	1134	74	447	2.93	0.22	0.78	3.93
Yylex (gcc)	1573	I	4632	3050	660	6.37	0.60	3.78	10.75
		II	1825	252	670	5.30	0.90	1.61	7.81
		III	1648	75	493	5.86	0.38	1.62	7.86
Reload (gcc)	344	I	1378	1034	410	3.56	0.15	0.87	4.58
		II	802	458	675	2.73	0.74	0.35	3.82
		III	525	181	385	3.15	0.25	0.27	3.67
Iscaptured (go)	61	I	194	133	55	0.12	0.01	0.04	0.17
		II	138	77	104	0.11	0.03	0.02	0.16
		III	89	28	57	0.12	0.02	0.02	0.16
Cse_insn (gcc)	396	I	1476	1080	367	3.05	0.18	0.69	3.92
		II	698	302	544	2.61	0.53	0.32	3.46
		III	492	96	344	2.85	0.18	0.27	3.30
Eval (perl)	1375	I	3224	1849	619	7.14	0.40	2.50	10.04
		II	1546	171	717	4.45	0.82	1.05	6.32
		III	1456	81	624	5.80	0.20	1.13	7.13
Ttin (o/s code)	539	I	2389	1850	826	3.44	0.20	1.41	5.05
		II	1369	830	1201	2.43	2.21	0.30	4.94
		III	761	222	600	2.59	0.87	0.18	3.64
load_data (m88ksim)	163	I	183	20	53	0.11	0.01	0.04	0.16
		II	163	0	53	0.10	0.01	0.03	0.14
		III	163	0	53	0.11	0.01	0.04	0.1
%improvement ⁸		I	*	*	*	*	*	*	*
		II	*	72.1	-29.1	*	*	*	13.1
		III	*	89.9	8.6	*	*	*	15.1

redundant copies that were present prior to the translation. Our experimental results

⁸

Percentage improvement is first calculated with respect to Method I for each procedure and is then averaged over all procedures.

corroborate this behavior (in many cases the AC value is less than the BT value for Method III). After copy elimination, Method II clearly has the most copies left, and Methods I and III have comparable numbers of copies left in most of the cases. But for procedures `yylex(gcc)`, `yylex(perl)`, and `ttn (o/s code)`. Method III is very effective in terms of reducing the number of copies in final code (an improvement of more than 24%). One reason for this dramatic improvement is because the coalescing algorithm has to eliminate more copies in Method I than in Method III. The copy elimination order is important for Method I since it impacts the total number of copies that are eliminated. Since Method III places only the copies that cannot be eliminated by coalescing, it is less affected by the elimination order. In summary, we conclude 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 the final code.

Next we will discuss the running time for all three methods. The column DFA/IG indicates the time for computing the data flow information and the interference graph. The column TT indicates the time for translating the TSSA form to the CSSA form, and TT also includes the time to incrementally update data flow information and interference graph for the relevant methods. Column TC indicates the time for our CSSA based coalescing. Finally, the last column indicates the summation of DFA/IG, TT, and TC.

From the table we can see that, for all three methods computing the data flow information and the interference graph dominates their overall running time. The time in DFA/IG tends to be longer for Method I because of a large number of extra copies inserted during TSSA-to-CSSA translation. Since Method III also tracks additional live-in sets, it takes more time than Method II for computing the data flow information. Method II needs to update data flow sets and interference graph for a relatively large number of copy instruction inserted during its TSSA-to-CSSA translation, so it usually requires the longest time under TT. Compared to the other two methods, the copy elimination in Method I has many more copy instructions to eliminate, and thus, Method I takes a longer time under TC.

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 average it is about 15% 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.

7. Discussion and Conclusion

In this paper we presented a new framework for translating out of SSA form and for eliminating redundant copies. Previous work that are most relevant to ours are due to Cytron et al. [7] and Briggs et al. [2]. Both methods pessimistically insert copies for all the source resources of a phi instruction. Our Method I and the previous two methods rely on a subsequent coalescing phase to eliminate redundant copies. Our experimental results indicate that pessimistically inserting copies increases the space/time requirements of the algorithm.

Cytron et al. never insert copies for the target resource of a phi instruction. The work in [2] showed that this leads to incorrect code generation in certain cases when a transformation, such as value numbering, is performed on the SSA form. Briggs et al.

[2] illustrated two examples, the “lost copy” problem and the “swap” problem, where the original Cytron et al. algorithm would fail [7]. They then proposed algorithms to handle both of these problems. Unlike our Method I, they judiciously place copies for the target resource of a phi instruction to handle the two special cases. The solution to the swap problem requires ordering on copy insertion, and the solution to the lost copy problem requires data flow live out information. To summarize their algorithm, basic blocks in the control flow graph of a program are visited in a preorder walk over the dominator tree. Each basic block is then iterated to replace uses in phi instructions and other instructions with any new names previously generated during the preorder walk. A list of copies that are needed in this basic block are built and inserted in the order determined by the algorithm that handles the swap problem. Finally, as each copy is inserted, an algorithm to handle the lost copy problem is invoked (which uses live out information to ensure that a needed copy is not lost).

We presented a uniform framework for eliminating phi instructions. Our framework is based on two important properties: (1) the phi congruence property, and (2) the property that none of the resources in a phi congruence class interfere. We use liveness analysis and an interference graph to eliminate phi instructions. In [2] the authors remark that they reduced the problem of eliminating phi instruction to a scheduling problem. In our framework we have reduced the problem of eliminating phi instructions to a coloring-based register allocation problem [3]. In this paper we presented one strategy for spilling copies, where copies for source resources of a phi instruction are inserted in the predecessor basic blocks, and a copy is inserted for the target resource in the same basic block as the phi instruction. One can envision other spilling strategies that can further reduce the number copies needed to correctly eliminate phi instructions. Note that, unlike the Briggs et al. algorithm, our framework does not use any structural properties of the control flow graph or the dependence graph induced by the SSA names (the SSA graph) to ensure that the copies are placed correctly. We also do not visit the basic blocks in any particular order to ensure that copies are placed correctly. Another unique aspect of our framework is the notion of phi congruence property of the CSSA form. We exploited this property to translate TSSA form to CSSA form by breaking interferences among resources in a phi congruence class, and then eliminating phi instructions in the CSSA form. Our new CSSA-based algorithm also uses phi congruence classes to correctly and aggressively eliminate copies.

Finally, it is important to note that although our framework handles all control flow structures including loops with multiple exits and irreducible control flow and does not require any explicit control flow structure for correctness, transformations, such as edge splitting, help reduce the number of copies that are inserted in our phi instruction elimination phase. It also helps remove more copies during our CSSA-based coalescing. We have also observed that visiting basic blocks in a certain order helps improve the effectiveness of both the phi instruction elimination phase and the copy elimination phase. These issues are part of future work and beyond the scope of this paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. P. Briggs, K. Cooper, T. Harvey, and Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software-Practice and Experience*, 28(8):859-881, July 1998.
3. G. J. Chaitin, Register allocation and spilling via graph coloring. *SIGPLAN Notices* 17(6):98-105, June 1982. Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction.
4. F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, "A New Algorithm for Partial Redundancy Elimination based on SSA Form," Proc. of the 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 273-286, 1997.
5. C. Click. Global Code Motion Global Value Numbering. *SIGPLA Notices*, 30(6):246-257, June 1995. Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation.
6. R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. *SIGPLAN Notices*, 28(6):36-45, June 1993. Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation
7. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.
8. A. Leung and L. George. Static Single Assignment Form for Machine Code. *SIGPLAN Notices*, 34(5):204-214, May 1999. Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation.
9. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, CA, 1997.
10. P. Pineo and M. L. Soffa. A practical approach to the symbolic debugging of parallelized code. Proceedings of International Conference on Compiler Construction, April 1994.
11. V. C. Sreedhar and G. R. Gao. Computing ϕ -nodes in linear time using DJ graphs. *Journal of Programming Languages*, 13(4):191-213, 1996.
12. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181-210, April 1991.

Appendix: 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  $x_0 = f(x_1:L1, x_2:L2, \dots, x_n:L_n)$ ;
   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 4.3.
        }
5:   Process the unresolved resources (Case 4) as described in Section 4.3.
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];
}
}
}

```