Journal of Universal Computer Science, vol. 12, no. 8 (2006), 1007-1019
submitted: 3/1/05, accepted: 25/8/06, appeared: 28/8/06 © *J.UCS*

Eliminating Redundant Join-Set Computations in Static Single Assignment

Angela French
(Department of Computing Science
University of Alberta, Edmonton, Canada
french@cs.ualberta.ca)

José Nelson Amaral
(Department of Computing Science
University of Alberta, Edmonton, Canada
amaral@cs.ualberta.ca)

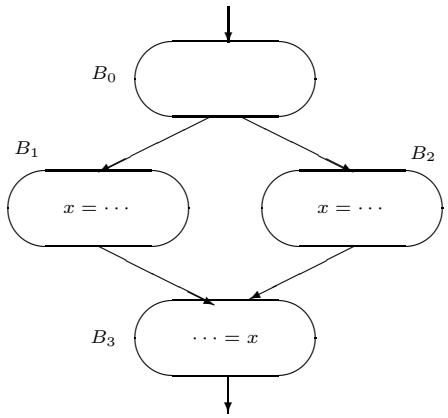
Abstract: The seminal algorithm developed by Ron Cytron, Jeanne Ferrante and colleagues in 1989 for the placement of ϕ-nodes in a control flow graph is still widely used in commercial compilers. Placing ϕ-nodes is necessary when converting a program representation to Static Single Assignment (SSA) form. This paper shows that if a variable x is defined in a set of basic blocks $A(x)$, then the iterated join set of $A(x)$ can be decomposed into the computation of the iterated join set of a disjoint collection of subsets of $A(x)$. We use this result to show that some join set computations are redundant. We measured the number of redundant computations in the Open Research Compiler (ORC) in a selection of SPEC 2000 benchmarks.

Key Words: compiler optimization, SSA
Category: D.1.3, D.3.4, I.1.2

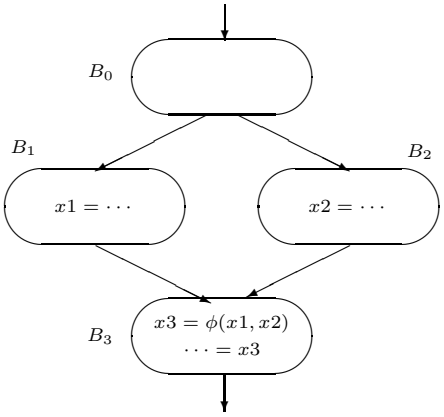
1 Introduction

Static Single Assignment (SSA) form has been used to facilitate the implementation of optimizations such as full and partial redundancy elimination, and constant propagation. The conversion of non-SSA code into SSA form has been extensively studied. The algorithm most commonly used in production compilers for this conversion into SSA is still the one described by Cytron *et al.* [9].

Given two distinct paths in a control flow graph, a join node is the first node, on a topological traversal, that belongs to the two paths. The conversion to SSA renames the variables in a program in such a way that in SSA each variable appears only once in the left hand side of any assignment. A difficulty appears when a variable x appears in the left hand side in distinct paths, and then it appears in the right hand side on or after a join node as illustrated by the



(a) Original Program.



(b) Program in SSA form.

Figure 1: Example of the use of ϕ-function in a join node.

example in Figure 1(a). The question is which value of x should be used on or after the join node. The solution is to add a statement at the join node that uses a special function called a ϕ-function [9]. Conceptually the ϕ-function "knows" which path the program took to get to the join node. Figure 1 illustrates the

transformation into SSA. The set of join nodes where ϕ-functions for a variable x must be inserted is called the join set of x, $J(X)$. The insertion of the additional assignment to x at a join node may itself cause new join nodes to be added to $J(X)$, therefore we must consider the *iterated join set* for x, $J^+(X)$, which is the limit of the sequence obtained by insertions of ϕ-functions for x in join nodes of x.

Citron *el al.*'s algorithm determines the set of Control Flow Graph (CFG) vertices where a ϕ-function must be inserted for a variable x using the iterated join set $J^+(X)$, where X is the set of vertices in the original CFG that contains a definition of x.[1] Computing the iterated join set for a variable x is expensive because it requires the computation of the iterated dominance frontier for the set of vertices that define x (see Theorem 1 in Section 3). A trivial observation is that if two variables x and y are defined in the same set of vertices, the vertices that require a ϕ-function for x and y are the same.

In this paper we revisit Cytron *et al.*'s ϕ-placement algorithm. Let $A(y)$ be the set of vertices that define y, and $A(x)$ be the set of vertices that define x. We show that if $A(y)$ is a subset of $A(x)$, then the iterated join set for y can be reused when computing the iterated join set for x. In order to show that reusing partial join sets is sound, we have to prove that if X is decomposed into disjoint multi subsets, $X = X_1 \cup X_2 \cup \ldots \cup X_n$, then the iterated join set of X is equal to the union of the iterated join sets of the subsets, $J^+(X) = J^+(X_1) \cup J^+(X_2) \cup \ldots \cup J^+(X_n)$.

An empirical study, presented in Section 5, finds dominance frontier computations that can be eliminated through the use of this theoretical result in a selection of SPEC 2000 CINT benchmarks. Eliminating dominance frontier computations is relevant because computing the dominance frontier is the most time consuming portion of the SSA conversion algorithm. The remainder of this paper is organized as follows. Section 2 presents an example to motivate the partitioning of join sets and the proofs presented in Section 3. The theoretical results are then used to argue, in Section 4, that some join set computations are redundant. Related work is discussed in Section 6.

2 Motivating Example

Consider the fragment of a control flow graph in Figure 2. The set of vertices that make an assignment to a variable x is called $A(x)$. In this example

[1] A vertex defines a variable x if x appears in the left-hand side of any statement in the vertex.

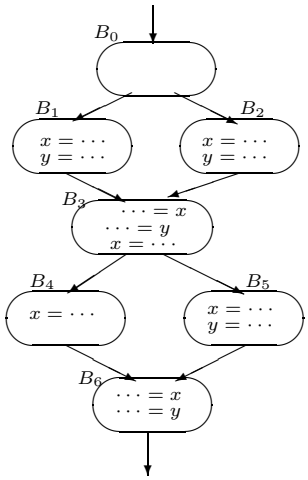


Figure 2: Example of sets of assignments for two variables, x and y

$A(x) = \{B_1, B_2, B_3, B_4, B_5\}$ and $A(y) = \{B_1, B_2, B_5\}$. Clearly, $A(y) \subset A(x)$. $A(x)$ can be split into two subsets: $A_1(x) = A(y) = \{B_1, B_2, B_5\}$ and $A_2(x) = A(x) - A(y) = \{B_3, B_4\}$. The iterated join sets of $A(y)$ and $A_1(x)$ are the same. The question is: can the iterated join set of $A(x)$, $J^+(A(x))$ be expressed as the union of the iterated join set of $A_1(x)$ and $A_2(x)$, or $J^+(A_1(x) \cup A_2(x))$? In Section 3 we show that the answer is yes. Therefore if $J^+(A(y))$ is computed first, then we only need to compute $J^+(A_2(x))$ to obtain $J^+(A(x))$. Cytron *et al.*'s algorithm is a worklist algorithm. Using the result of this paper, the size of this worklist can be reduced, thus resulting in fewer computations.

3 Join-Set Partitioning

The objective of this paper is to improve the conversion of intermediate code to static single-assignment form within a procedure. The notation $G(V, E)$ refers to a CFG whose vertices V represent the procedure's basic blocks. An edge $(B_i, B_j) \in E$ represents the possibility that control may be transfered from B_i to B_j at runtime. A time-intensive phase of an SSA conversion algorithm is the determination the minimum set of nodes that requires a ϕ-function for each variable x in the code, or $S_\phi(x)$. A variable that is defined more than once in the original code requires a ϕ-function in each node joining paths that contain definitions of x. The problem is compounded by the fact that a ϕ-function is

itself a definition of x. Therefore the algorithm that computes the join set for a variable has to iterate until no more join nodes are found.

Definition 1 Given a control flow graph $G(V, E)$ with a distinct start vertex $B_s \in V$. If every path from B_s to $B_j \in V$ goes through $B_i \in V$, then B_i *dominates* of B_j. Every vertex dominates itself. If B_i dominates B_j and $B_i \neq B_j$, then Bi *strictly dominates* B_j.

Definition 2 Consider a node $B_i \in V$. The *dominance frontier* of B_i, $DF(B_i)$, is the set of all nodes $B_j \in V$ such that B_i dominates a predecessor of B_j, but B_i does not strictly dominate B_j itself [9].

When constructing the SSA form, if a variable x has multiple definitions, the SSA-conversion algorithm determines all join nodes where a ϕ-function for x has to be inserted in a single pass. Therefore sets of nodes have to be analysed [9]. Let X be a set of CFG nodes that contain definitions of a variable x. Then,

$$DF(X) = \bigcup_{B_i \in X} DF(B_i) \tag{1}$$

Definition 3 Given a set of CFG nodes X, the *iterated dominance frontier* of X, $DF^+(X)$, is the limit of the following sequence [9]*:*

$$DF_1 = DF(X) \tag{2}$$

$$DF_{i+1} = DF(X \cup DF_i) \tag{3}$$

The primary ϕ-placement method still used in many compilers was presented by Cytron *et al.* in 1989 [9], and further elaborated on in 1991 [10]. This method uses dominance frontiers to determine where the ϕ-functions should be placed. The relationship between dominance frontiers and ϕ-functions is established by Theorem 1 [10].

Theorem 1 *The set of nodes that need ϕ-functions for any variable x is the iterated dominance frontier $DF^+(A(x))$. Equivalently,*

$$J^+(A(x)) = DF^+(A(x)) \tag{4}$$

Our goal is to prove that if $X = X_1 \cup X_2$, then the join set of X is the union of the join set of X_1 and the join set of X_2 (see Theorem 4). However, some preliminary results are necessary. Theorem 2 establishes that if the subsets form a partition of X, then the dominance frontier of the union is equal to the union of the dominance frontiers. Theorem 3 establishes a similar result for the iterated dominance frontier.

Theorem 2 *Let X be a subset of nodes in V such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$DF(X_1) \cup DF(X_2) = DF(X_1 \cup X_2) \tag{5}$$

Proof: By Equation 1, we know that:

$$DF(X) = \bigcup_{B_i \in X} DF(B_i)$$

Thus,

$$\begin{aligned} DF(X_1) \cup DF(X_2) &= \bigcup_{B_i \in X_1} DF(B_i) \cup \bigcup_{B_i \in X_2} DF(B_i) \\ &= \bigcup_{B_i \in (X_1 \cup X_2)} DF(B_i) \\ &= DF(X_1 \cup X_2) \end{aligned}$$

□

Theorem 3 *Let X be a subset of nodes in V such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$DF_i(X_1) \cup DF_i(X_2) = DF_i(X_1 \cup X_2) \tag{6}$$

where DF_i is an element of the sequence that defines DF^+.

Proof: The proof is by induction.
Base case: If $i = 1$, from Definition 3 we know that $DF_1 = DF(X)$. Hence:

$$DF_1(X_1) \cup DF_1(X_2) = DF(X_1) \cup DF(X_2)$$

From Theorem 2, we have:

$$DF(X_1) \cup DF(X_2) = DF(X_1 \cup X_2)$$

Thus,

$$DF_1(X_1) \cup DF_1(X_2) = DF(X_1 \cup X_2) = DF_1(X_1 \cup X_2)$$

Inductive case: Assume that $DF_i(X_1) \cup DF_i(X_2) = DF_i(X_1 \cup X_2)$ for $i = k$. Let $i = k + 1$. From Definition 3 we know that $DF_{i+1}(X) = DF(X \cup DF_i(X))$.

Then:

$$\begin{aligned} DF_{k+1}(X_1) \cup DF_{k+1}(X_2) &= DF(X_1 \cup DF_k(X_1)) \cup DF(X_2 \cup DF_k(X_2)) \\ &= DF(X_1) \cup DF(DF_k(X_1)) \\ &\quad \cup DF(X_2) \cup DF(DF_k(X_2)) \\ &= DF(X_1 \cup X_2) \cup DF(DF_k(X_1 \cup X_2)) \\ &= DF(X_1 \cup X_2 \cup DF_k(X_1 \cup X_2) \\ &= DF_{k+1}(X_1 \cup X_2) \end{aligned}$$

□

Theorem 4 *Let X be a subset of nodes in V such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$J^+(X) = J^+(X_1) \cup J^+(X_2) \tag{7}$$

Proof: Since G is a finite graph, $DF^+(X)$ must be finite. We know from Definition 3 that $DF^+(X)$ is the limit of a sequence of elements $DF_i(X)$. Equivalently,

$$DF^+(X) = \lim_{i \to c} DF_i(X)$$

where c is a constant. Recall from Theorem 1 that $DF^+(X) = J^+(X)$. Now

$$\begin{aligned} J^+(X_1) \cup J^+(X_2) &= DF^+(X_1) \cup DF^+(X_2) \\ &= \lim_{i \to c} DF_i(X_1) \cup \lim_{i \to c} DF_i(X_2) \\ &= \lim_{i \to c} [DF_i(X_1) \cup DF_i(X_2)] \\ &= \lim_{i \to c} DF_i(X_1 \cup X_2) \\ &= \lim_{i \to c} DF_i(X) \\ &= DF^+(X) \\ &= J^+(X) \end{aligned}$$

□

The next section shows that the result of Theorem 4 can be used in practice to eliminate the computation of redundant iterated dominance frontiers and iterated join sets.

4 Some Join-Set Computations are Redundant

Let x and y be variables in a program. Let $A(x)$ and $A(y)$ be the set of all vertices in V that contain a definition of x and y, respectively. If $A(x) = A(y)$, then $J^+(A(x)) = J^+(A(y))$. And we only need to compute one join set for both variables.

Moreover, if $A(y) \subset A(x)$, then $J^+(A(y)) \subset J^+(A(x))$. Recall that $J^+(A(x)) = S_\phi(x)$, the minimum set of nodes where ϕ-functions are required when constructing the SSA form of a program. If one set of assignment nodes is a subset of another, two join set computations are still required. The intersection of the two sets (*i.e.*, the smaller set) will be calculated. However, the second computation (*i.e.*, the remainder of the larger set) will be smaller than the original.

The majority of the time in Cytron *et al.*'s worklist algorithm is spent iterating over the worklist W, and every variable has a worklist associated with it. In particular, the worklist for variable v is initialized to $A(v)$. Then, each element $B_i \in W$ is removed from W, and a ϕ-function is inserted in every $B_j \in DF(B_i)$. As well, each B_j that now contains a ϕ-function is also inserted in W, and the process continues.

Consider one approach for the case $A(y) \subset A(x)$, where $A(x)$ is split into two smaller sets, $A(x_1) = A(y)$ and $A(x_2) = A(x) - A(y)$. This set division is possible since Theorem 4 determined that $J^+(A(x)) = J^+(A(x_1)) \cup J^+(A(x_2))$. Processing for $A(x_1)$ can be performed as normal. Consider here $A(x_2)$. The list of nodes that require a ϕ-function because of $A(x_2)$ will have to be computed separately as a worklist for x. However, savings still exists, as $A(x_1)$ and $A(y)$ are combined, and the worklist for $A(x_2)$ requires fewer iterations since $A(x_2) < A(x)$.

5 Saving Oportunities in Selected Benchmarks

An initial analysis of individual SPEC CINT2000 benchmarks [8] reveals the opportunities for join set computation optimization. All measurements in this section were performed in the Open Research Compiler for SSA translation level 1 (refer to Table 1) and optimization level O2 for baseline results [14].

The second column of Table 2 presents the number of worklists processed for the conversion into SSA of each one of the benchmarks. The column labeled $A(x) \subseteq A(y)$ presents the number of variable pairs x and y for which we can use the results presented in this paper to avoid redundant join set computations.

Translation level	Where the SSA removal is performed
1	after extended block optimizer preprocessing
2	after first pass of control flow optimization
3	after if-conversion
4	after second pass of control flow optimization
5	after extended block optimization
6	after global scheduling

Table 1: Description of SSA translation levels in the Open Research Compiler (ORC)

Benchmark	Total number of worklists processed	$A(x) \subseteq A(y)$	Percentage of work saved	$A(x) = A(y)$	Percentage of worklist avoided
164.gzip	2787	220	7.89	89	3.19
181.mcf	246	51	20.73	22	8.94
197.parser	17567	698	3.97	526	2.99
254.gap	115686	3555	3.07	1928	1.67
255.vortex	37372	1294	3.46	751	2.01
256.bzip2	12089	348	2.88	177	1.46
Average	30958	1028	3.32	582	1.88

Table 2: Opportunities for eliminating redundant join set computations in SPEC CINT2000 benchmarks

The percentage of work saved is the number of worklist computations that will be either avoided or reduced because of this finding.

The number of variable pairs for which $A(x) = A(y)$ is also shown in Table 2. The numbers reported in this column are also included in the $A(x) \subseteq A(y)$ column. The last column of Table 2 shows the percentage of worklists that can be eliminated altogether because of identical assignment sets. In this situation the worklist algorithm only needs to iterate for one of the two variables, $W = A(x) = A(y)$. For every $B_i \in W$ and $B_j \in DF(B_i)$, two ϕ-functions are added in B_j, one each for x and y. Performing the ϕ-function insertion in this manner eliminates an entire worklist iteration, and thus an entire join set computation, along with cutting down on accesses to the dominance frontier data structure. In practice, ϕ-functions are placed via a function call in the modified

Benchmark	$A(x) \subset A(y)$	Average difference in size of subsets	Average percentage saved
164.gzip	131	6.79	38.38
181.mcf	29	3.45	42.99
197.parser	172	2.65	52.10
254.gap	1627	4.34	45.35
255.vortex	543	5.00	53.14
256.bzip2	171	5.65	41.33
Average	445.5	5.58	45.55

Table 3: Instances where one join set is a subset of another join set in SPEC CINT2000 benchmarks

ORC SSA implementation in the code generator. Therefore, further savings can be achieved by removing that function call.

Table 3 gives the number of instances for the case $A(y) \subset A(x)$, along with the average difference between the size of the sets. This case is less attractive than the $A(x) = A(y)$ situation since it requires more work. However, if $A(x_2)$ is much smaller than $A(x)$, then the benefits could be significant. The third column of Table 3 indicates that the difference between the set sizes is small, and an average savings of 45% of the worklist computations, for these cases, can be realized.

Checking the relationships between these sets requires some extra calculation, but much of the work is facilitated through existing data structures in the SSA code. The one-time expense incurred to build correspondences between individual sets should be worth the benefits achieved through minimizing join set computations.

6 Related Work

The theoretical results presented in Section 3 were first published in French's M.Sc. thesis [12].

Cytron and Ferrante developed an algorithm to produce a *pruned* SSA form [7]. In pruned SSA the number of ϕ-functions placed is reduced because a ϕ-function for x is placed at a join node only if x is used within or after the join node, *i.e.*, x is live at the entry point of the join node. In [11], Cytron and Ferrante present a method that avoids computing all the dominance frontiers by ordering the

dominance frontier relation based on the position of the nodes in the dominator tree.

Briggs *et al.* [3] developed a *semi-pruned SSA* form that takes advantage of the observation that the life range of many names in a program is restricted to a single basic block. Thus, by computing the set of names that are live-in to some basic block, they are able to simplify the live analysis required for the computation of pruned SSA. Using this technique will result in a better trade-off between the precision of the ϕ-function placement and the time required for the analysis.

Sreedhar and Gao defined a ϕ-placement algorithm that does not require the computation of individual dominance frontiers [15]. This algorithm uses a *DJ-graph* which is a modification of the traditional dominator tree to compute the dominance frontiers on an as-needed basis.

Bilardi and Pingali's ϕ-placement algorithm was introduced in 1995 [13] and revisited in 2003 in an extensive comparative study of SSA construction techniques [1, 2]. This algorithm uses an *augmented dominator tree* (ADT) with a parameter β to control a space-time trade-off. Intuitively the trade-off is between precomputing all dominance frontiers, precomputing some, or computing them only on-demand. This ADT algorithm should subsume both Cytron *et al.*'s and Sreedhar-Gao's algorithms because they can be described as ADT with a fixed β value.

Several techniques have been proposed for translating out of SSA form. This translation requires the insertion of copies to ensure correctness when the ϕ-functions are removed. The problem can be formulated as minimizing the number of SSA-translation copy instructions executed at runtime. Sreedhar *et al.* present three methods to compute this translation [16]. Budimlić *et al.* perform copy folding during the translation to eliminate the need for a separate copy coalescing phase [4].

7 Conclusion

This paper presents profs that given two disjoint set of vertices in a control flow graph, $A_1(x)$ and $A_2(x)$, the iterated join set of their union, $J^+(A_1(x) \cup A_2(x))$, is equal the union of the individual iterated join sets, $J^+(A_1(x)) \cup J^+(A_2(x))$. This is a general flow analysis result. Then it showed that this property can be used to eliminate redundant iterated join set computations in a conversion to SSA algorithm. Finally it presented evidence that opportunities to realize such savings do indeed appear in industry-standard benchmarks in a robust compiler.

The contribution made in this paper is relevant because Cytron and Ferrante's algorithm is implemented in several commercial compilers. The modification required to obtain savings is trivial, and industry-standard benchmarks present opportunities to realize computational savings. Moreover similar savings should be realizable in recent SSAs formulations for predicated programs [5, 6, 17].

Acknowledgements

To perform this research we used the Open Research Compiler (ORC) infrastructure. Therefore we are standing on the shoulders of many compiler researchers and practitioners that developed this infrastructure over many years. We are grateful to all of them. Moreover we have special thanks for Arthur Stoutchinin for the many discussions on SSA generation. This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC). We thank anonymous referees for their insightful comments and suggestions.

References

1. B. Bilardi and K. Pingali. The static single assignment form and its computation. Technical report, Cornell University, 1999.
2. G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, 2003.
3. P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software-Practice and Experience*, 28(8):859–881, July 1998.
4. Z. Budimlić, K.D. Cooper, T.J. Harvey, K. Kennedy, T.S. Oberg, and S.W. Reeves. Fast copy coalescing and live-range identification. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 37 of *ACM SIGPLAN Notices*, pages 25–32, May 2002.
5. L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 245–255, Oct. 1999.
6. L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.
7. J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Conference on Principles of Programming Languages*, pages 55–66, 1991.
8. The Standard Performance Evaluation Corporation. *The SPEC CINT2000 Benchmark Suite*. http://www.spec.org/cpu2000/, 2003.
9. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

10. 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, Oct. 1991.
11. R.K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995.
12. Angela French. A study of later phase static single assignment in the open research compiler. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, AB, Canada, December 2003.
13. K. Pingali and G. Bilardi. APT: A data structure for optimal control dependence computation. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 32–46, June 1995.
14. SourceForge. *Open Research Compiler*. http://ipf-orc.sourceforge.net/, 2003.
15. V.C. Sreedhar and G.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, 1995.
16. V.C. Sreedhar, R.D.-C. Ju, D.M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Static Analysis 1999*, volume 1694 of *Lecture Notes in Computer Science*, pages 194–210.
17. A. Stoutchinin and F. de Ferriere. Efficient static single assignment form for predication. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 172–181, Dec. 2001.