Simple Generation of Static Single-Assignment Form

John Aycock and Nigel Horspool

Department of Computer Science,
University of Victoria,
Victoria, B. C., Canada V8W 3P6
{aycock,nigelh}@csc.uvic.ca

Abstract. The static single-assignment (SSA) form of a program provides data flow information in a form which makes some compiler optimizations easy to perform. In this paper we present a new, simple method for converting to SSA form, which produces correct solutions for nonreducible control-flow graphs, and produces minimal solutions for reducible ones. Our timing results show that, despite its simplicity, our algorithm is competitive with more established techniques.

1 Introduction

The static single-assignment (SSA) form is a program representation in which variables are split into "instances." Every new assignment to a variable — or more generally, every new definition of a variable — results in a new instance. The variable instances are numbered so that each use of a variable may be easily linked back to a single definition point.

Figure 1 gives a example of SSA form for some straight-line code. As its name suggests, SSA only reflects static properties; in the example, V_1's value is a dynamic property, but the static property that all instances labelled V_1 refer to the same value will still hold.

`read V`	$\Longrightarrow$	`read` $\mathtt{V}_1$
`V` $\leftarrow$ `V + 1`		$\mathtt{V}_2 \leftarrow \mathtt{V}_1$ `+ 1`

Fig. 1. SSA form of some straight-line code.

A problem arises at join points, where two or more control-flow paths merge. Multiple definitions of a variable may reach the join point; this would result in a violation of the single-assignment property. The problem is illustrated in Fig. 2a; what definition should the final instance of V be associated with?

To work around this problem, imaginary assignments are introduced at join points from trivial ϕ-functions. A ϕ-function has one argument for each incoming

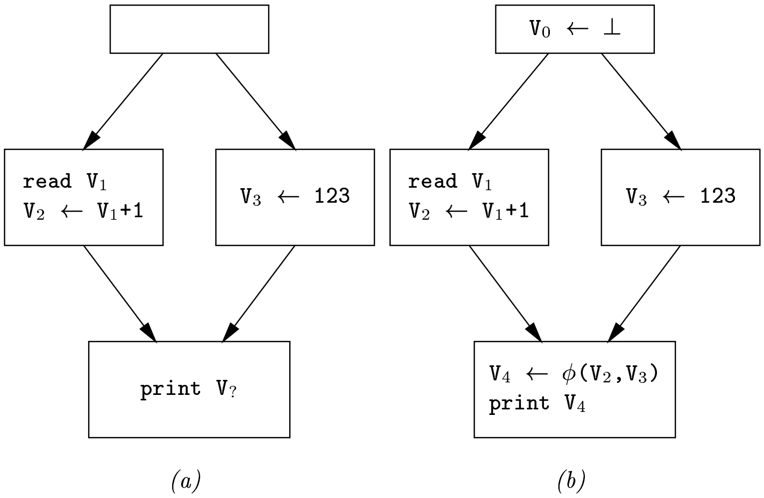


Fig. 2. Why ϕ?

control-flow path; the kth argument to a ϕ-function is the incoming value along the kth path.

Figure 2b shows the inserted ϕ-function. An initial assignment to V has been added so that instances of V always have a corresponding definition.

ϕ-functions are always inserted at the beginning of a basic block, and are considered to be executed simultaneously before execution of any other code in the block. A program must be converted out of SSA form before it is executed on a real machine.

Why use SSA form? Proponents of SSA have cited many advantages:

1. Every use of a variable is dominated by a definition of that variable [2, 4]. Some optimization algorithms may be made more efficient by taking advantage of this property [4, 21].
2. SSA chains are simpler to store and update than use-def chains [10].
3. Use-def chains may cause some optimizations to be missed that would be caught with a SSA-based algorithm [27].
4. Distinct uses of a variable in the source program — reusing a loop variable for another purpose, for example — become distinct variables in SSA form. This may allow more optimizations to be performed [3].

2 Converting to SSA Form

SSA form, and the conversion to SSA form, is closely coupled to dominance. If a definition of a variable V dominates a use of V, then that use may be linked back to a single definition. At join points, several definitions of V may reach a use of V, and so a ϕ-function is needed. Where ϕ-functions are needed, then, is where a definition *stops* dominating: this is the dominance frontier.

This is the basis of the algorithm by Cytron et al. [10], which is by far the most often-cited method for converting into SSA form. The idea is to precompute the dominance frontiers, then use that information to place a minimal number of ϕ-functions. The ϕ-functions are themselves definitions with a dominance frontier, so the process must be repeated — ϕ-functions must be placed in all basic blocks in the *iterated* dominance frontier.

The argument has been made that only the minimal number of ϕ-functions required should be inserted; otherwise, some optimizations could be missed [10, 27]. While other forms of "minimal" SSA exist, such as those those taking liveness of variables into account, we do not consider them here.

3 Converting to SSA Form, Revisited

Appel [3] gives a gentle introduction to SSA form. He begins by suggesting a wasteful but obviously correct method for converting to SSA:

> 'A *really crude approach* is to split every variable at every basic-block boundary, and put ϕ-functions for every variable in every block.' [3, page 17]

He then recounts the dominance frontier algorithm of Cytron et al. [10] which inserts a minimal number of ϕ-functions. Appel's presentation raises the question: could a minimal number of ϕ-functions be discovered by starting with the "really crude approach" and iteratively deleting extraneous ϕ-functions?

3.1 Our Algorithm

Our algorithm finds a set of ϕ-functions for a given variable in a reducible control-flow graph. Intuitively, a reducible control-flow graph is one which does not have multiple entries into a loop. This is an important class of control-flow graphs because many modern languages, such as Oberon and Java, only admit reducible control-flow graphs; there is also empirical evidence suggesting that people tend to write programs with reducible control-flow graphs even if they can do otherwise [17].

We assume that there are no unreachable nodes in the control-flow graph, although the algorithm will still derive a correct solution in this case.

Our algorithm proceeds in two phases:

RC phase. Apply Appel's "really crude" approach as quoted above.
Minimization phase. Delete ϕ-functions of the form

$$V_i \leftarrow \phi(V_i, V_i, \ldots, V_i)$$

and delete ϕ-functions of the form

$$V_i \leftarrow \phi(V_{x_1}, V_{x_2}, \ldots, V_{x_k}), \text{where } x_1, \ldots, x_k \in \{i, j\}$$

replacing all other occurrences of V_i with V_j. Repeat this until no further minimizations are possible.

Once the above phases have determined the set of ϕ-functions to insert, then another pass over the control-flow graph renames instances of variables to their SSA forms. This is not unique to our algorithm, and will not be mentioned further.

An example is shown in Figs. 3–6. The original program is listed in Fig. 3a; the result after the RC phase is Fig. 3b. Figures 4 and 5 show the sets of ϕ-functions converging for `i` and `j`, respectively, and Fig. 6 gives the final result. Normally, the program variables would not be renamed until after the minimization phase, but they have been renamed earlier for illustrative purposes.

```
i ← 123
j ← i * j
repeat
    write j
    if (j > 5) then
        i ← i + 1
    else
        break
    end
until (i > 234)
```

(a)

```
i0 ← ⊥
j0 ← ⊥
i1 ← 123
j1 ← i1 * j0
repeat
    i2 ← φ(i1, i6)
    j2 ← φ(j1, j5)
    write j2
    if (j2 > 5) then
        i3 ← φ(i2)
        j3 ← φ(j2)
        i4 ← i3 + 1
    else
        i5 ← φ(i2)
        j4 ← φ(j2)
        break
    end
    i6 ← φ(i4)
    j5 ← φ(j3)
until (i6 > 234)
i7 ← φ(i6, i5)
j6 ← φ(j5, j4)
```

(b)

Fig. 3. Before and after the RC phase.

3.2 Correctness

In this section we prove the correctness of the algorithm. By "correct," we mean that our algorithm always produces a set of ϕ-insertions that is a (possibly improper) superset of the minimal solution. Note that nothing in this proof

$$
\begin{array}{l}
i_2 \leftarrow \phi(i_1, i_6) \\
i_3 \leftarrow \phi(i_2) \\
i_5 \leftarrow \phi(i_2) \\
i_6 \leftarrow \phi(i_4) \\
i_7 \leftarrow \phi(i_6, i_5)
\end{array}
\quad
\overset{\Longrightarrow}{\begin{bmatrix} i_3 \equiv i_2 \\ i_5 \equiv i_2 \\ i_6 \equiv i_4 \end{bmatrix}}
\quad
\begin{array}{l}
i_2 \leftarrow \phi(i_1, i_4) \\
i_7 \leftarrow \phi(i_4, i_2)
\end{array}
$$

Fig. 4. Minimization phase convergence for `i`.

$$
\begin{array}{l}
j_2 \leftarrow \phi(j_1, j_5) \\
j_3 \leftarrow \phi(j_2) \\
j_4 \leftarrow \phi(j_2) \\
j_5 \leftarrow \phi(j_3) \\
j_6 \leftarrow \phi(j_5, j_4)
\end{array}
\quad
\overset{\Longrightarrow}{\begin{bmatrix} j_3 \equiv j_2 \\ j_4 \equiv j_2 \\ j_5 \equiv j_4 \equiv j_2 \\ j_6 \equiv j_2 \end{bmatrix}}
\quad
j_2 \leftarrow \phi(j_1, j_2)
\quad
\overset{\Longrightarrow}{\begin{bmatrix} j_2 \equiv j_1 \end{bmatrix}}
\quad
\text{(none)}
$$

Fig. 5. Minimization phase convergence for `j`.

```
i_0 ← ⊥
j_0 ← ⊥
i_1 ← 123
j_1 ← i_1 * j_0
repeat
    i_2 ← φ(i_1, i_4)
    write j_1
    if (j_1 > 5) then
        i_4 ← i_2 + 1
    else
        break
    end
until (i_4 > 234)
i_7 ← φ(i_4, i_2)
```

Fig. 6. After the minimization phase (and renaming).

requires the control-flow graph to be reducible, so our algorithm produces a correct, but not necessarily minimal, solution for nonreducible graphs as well.

Lemma 1. *The RC phase produces a correct solution.*

Proof. Since the RC phase places ϕ-functions in all basic blocks, the minimal placement of ϕ-functions must be contained within the initial placement upon completion of the RC phase. □

Lemma 2. *The minimization phase produces a correct solution.*

Proof. There are two transformations performed in this phase:

1. Deleting $V_i \leftarrow \phi(V_i, V_i, \ldots, V_i)$. This can be safely deleted because it corresponds to the assignment $V_i \leftarrow V_i$ on all predecessor edges,[1] which has no effect on the program state. A minimal solution could not contain this because it is clearly superfluous.
2. Deleting $V_i \leftarrow \phi(V_{x_1}, V_{x_2}, \ldots, V_{x_k}), x_1, \ldots, x_k \in \{i, j\}$, and replacing all other occurrences of V_i with V_j. The ϕ-function assignment corresponds to the set of assignments $\{V_i \leftarrow V_i, V_i \leftarrow V_j\}$. As before, $V_i \leftarrow V_i$ has no effect and can be ignored. In the case of $V_i \leftarrow V_j$, it means that V_i must have the value V_j at all points in the program due to the single-assignment property. It is therefore safe to replace all V_i with V_j. Since V_i's only rôle is as an alternate name for V_j, it could not be part of a minimal solution.

□

Theorem 1. *Our algorithm produces a correct solution.*

Proof. By Lemmas 1 and 2, our algorithm cannot remove needed ϕ-functions, and must arrive at a (possibly improper) superset of the minimal solution. □

3.3 Proof of Minimality

In this section we prove that, for reducible control-flow graphs, our algorithm produces a minimal placement of ϕ-functions.

This proof draws from T_1-T_2 reduction of control-flow graphs [14]. To briefly summarize T_1-T_2 reduction, transformation T_1 is removal of a self-edge; transformation T_2 allows a node n_1 to eliminate a node n_2 if n_1 is the unique predecessor of n_2, and n_2 is not the initial node.

We construct the instance relationship graph, or IR-graph, as a directed graph derived from the control-flow graph that shows the relationships between instances of a variable in SSA form.[2] Every variable gives rise to a different IR-graph. Each instance V_i of a variable becomes a node in the IR-graph; for each

[1] These are placed on edges for the purposes of this proof, to avoid the critical edge problem cited by [6, 20].

[2] The IR-graph is only used for the purposes of this proof; it is not used by our algorithm.

ϕ-function $V_i \leftarrow \phi(V_{x_1}, V_{x_2}, \ldots, V_{x_k})$, we add k edges to the IR-graph:

$$\begin{array}{c} V_{x_1} \rightarrow V_i \\ V_{x_2} \rightarrow V_i \\ \vdots \\ V_{x_k} \rightarrow V_i \end{array}$$

Not all instances are defined in terms of ϕ-functions. We call definitions that do not correspond to left-hand sides of ϕ-function assignments "real definitions." An instance V_i which corresponds to a real definition requires some special attention:

Case 1. V_i is not live[3] across a basic block boundary. This case corresponds to a temporary instance of V created and killed within a single basic block B. V_i will not appear in any ϕ-function. (The *final* definition of V in B will be cited by ϕ-functions in B's successor blocks.) V_i will appear as a disconnected node in the IR-graph and may be deleted; it is not taken into account by our algorithm since it doesn't appear in any ϕ-function. V_i is irrelevant to a ϕ-placement algorithm based on dominance frontiers too, because the definition giving rise to V_i in B only dominates a set of successor instructions in B, so B doesn't appear in any dominance frontier as a result of the definition.

Case 2. V_i is live across a basic block boundary. V_i must then appear as an argument to at least one ϕ-function. In the IR-graph, the V_i node will appear to be a "root" of the graph, since it will have an in-degree of zero. Let R be the set of roots of the IR-graph. So that we may take advantage of graph reduction techniques, we always augment the IR-graph with a "supersource" node V_S [13], which becomes the root of the IR-graph, and has an edge from it to every element of R. The supersource will be shown later to be inert with respect to the proof.

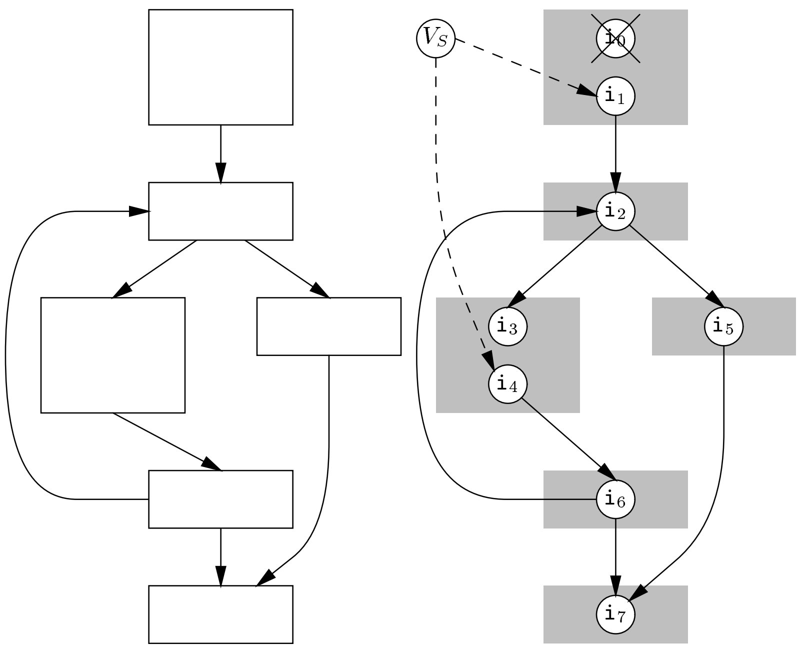
Case 3. V_i's definition reaches the exit point of the control-flow graph, but does not cross any basic block boundaries. As in Case 1, V_i will not appear in any ϕ-function, will appear as a disconnected node in the IR-graph, and may be deleted.

After the RC phase, when all basic blocks contain a ϕ-function, the IR-graph's structure will ape the structure of the control-flow graph. For example, Fig. 7a shows the control-flow graph for the code in Fig. 3a; the corresponding IR-graph for `i` is given in Fig. 7b.

In the IR-graph, the nature of the minimization phase is now apparent:

1. Deleting $V_i \leftarrow \phi(V_i, V_i, \ldots, V_i)$ is equivalent to applying T_1 to the IR-graph.
2. Deleting $V_i \leftarrow \phi(V_{x_1}, V_{x_2}, \ldots, V_{x_k}), x_1, \ldots, x_k \in \{i, j\}$, and replacing all other occurrences of V_i with V_j is equivalent to applying T_2 (possibly in combination with T_1) to the IR-graph.

[3] We consider liveness to include uses of V_i as arguments to ϕ-functions.



(a) *(b)*

Fig. 7. Control-flow and IR graphs.

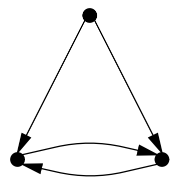


Fig. 8. The (*)-graph.

The structure of the IR-graph is important too:

Lemma 3. *The IR-graph is nonreducible only if the control-flow graph is nonreducible.*

Proof. Assume that the control-flow graph is reducible and the IR-graph is nonreducible. Then the IR-graph must contain the (*)-graph [14], which is illustrated in Fig. 8. Each edge in the IR-graph arises in one of two ways:

1. The edge results from paths in the control-flow graph. If every edge in the (*)-graph came from the control-flow graph, then the control-flow graph was itself nonreducible [14], yielding a contradiction.
2. The edge is added from the supersource V_S. V_S has no in-edges, so if any edge from V_S were to form part of the (*)-graph, it would have to connect to a V_i node which is part of a cycle in the graph. However, by definition of the IR-graph construction, the V_i nodes that V_S would connect to have no other in-edges, thus they cannot be part of a cycle.

□

Since we are only considering reducible control-flow graphs, the IR-graph cannot initially contain the (*)-graph by Lemma 3. Furthermore, the (*)-graph cannot be introduced through T_1-T_2 reduction [14]. This means that the IR-graph must be reducible by T_1-T_2 transformations into the single node V_S.

T_1 and T_2 comprise a finite Church-Rosser transformation [14]. This means that if T_1 and T_2 are applied to a graph until no more transformations are possible, then a unique graph will result [1] — in this case, the single node V_S. Furthermore, this unique result does not depend on the order in which T_1 and T_2 are applied [13].

Given this freedom, we choose an ordering of T_1-T_2 reductions which corresponds to the manner in which our algorithm operates. A parse of a reducible flow graph is an ordered sequence of reductions together with the nodes to which the reductions are applied [13, 14]. We select a full parse of the IR-graph which may be partitioned in two:

1. The first part performs as many T_1 and T_2 reductions as possible without eliminating any root nodes in R.
2. The final part applies T_1 and T_2 transforms to the remainder of the IR-graph, reducing it into the single node V_S. $\Pi(R)$ refers to the set of nodes yet to be reduced in the latter partition; by definition, $R \subseteq \Pi(R)$. $V_S \notin \Pi(R)$ because V_S can never be eliminated.

Lemma 4. *The minimization phase computes $\Pi(R) - R$.*

Proof. The transformations performed in the minimization phase can only remove V_i instances resulting from ϕ-functions; they cannot remove V_i instances corresponding to real definitions. This is the same as applying T_2 to the IR-graph, subject to the proviso that no elements of R be deleted. Because the minimization phase repeats until no further transformations are possible, it is computing $\Pi(R) - R$. □

Lemma 5. *$\Pi(R) - R$ is the iterated dominance frontier of R, $DF^+(R)$.*

Proof. The RC phase inserts a ϕ-function per basic block, so every V_i resulting from a ϕ-function has a one-to-one correspondence to a node in the control-flow graph. By definition, $\Pi(R) - R$ cannot contain any V_i from real definitions, so we may discuss the IR-graph nodes in $\Pi(R) - R$ and the control-flow graph nodes in $DF^+(R)$ interchangeably.

$\Pi(R) - R \subseteq DF^+(R)$. A reducible flow graph can be thought of as being decomposed by "regions." [26] A region is a subflowgraph, and the header node of a region dominates all nodes in the region save itself [14]. When the regions which have the elements of R as their headers are eliminated via T_1 and T_2, then what remains is the set of nodes which are not strictly dominated by elements of R. In other words, we are left with the dominance frontier of R, $DF(R)$.

The nodes in $DF(R)$ will themselves be headers of regions which have been reduced via T_1 and T_2. Inductively repeating this process, we get the iterated dominance frontier of R.

$DF^+(R) \subseteq \Pi(R) - R$. Suppose that there were a basic block $B \in DF^+(R)$ such that its corresponding IR-graph node $V_B \notin \Pi(R) - R$. This means that V_B must have already been eliminated by T_1 and T_2 earlier in the reduction parse. For this to happen, V_B must have been strictly dominated by some node in the IR-graph. It could not then be in $DF^+(R)$, a contradiction. □

Theorem 2. *Our algorithm computes the minimal ϕ-function placement for reducible control-flow graphs.*

Proof. By Lemmas 4 and 5, our algorithm computes the iterated dominance frontier of R, where R is the set of real definitions of V. This iterated dominance frontier has been shown to be the minimal ϕ-function placement [10]. □

3.4 Improvements to the Basic Algorithm

Our algorithm can be improved upon; three improvements are immediately apparent:

Improvement 1. *One-pass RC phase.*

When inserting ϕ-functions during the RC phase, the instances of a variable coming from predecessor blocks must be known; complete processing of a block requires that all of its predecessor blocks be processed first. Even the best case — a depth-first ordering of the blocks — may require backpatching of information along back edges.

A slight change in numbering fixes this. If the instance of a variable V coming out of a block B_i is always V_i, then a block may be completely processed simply by knowing the block numbers of its predecessors — information likely to be available anyway. This means that the RC phase can run linearly regardless of how the blocks are ordered.

Improvement 2. *Mapping table.*

A naïve implementation of the minimization phase, which literally renamed all instances of V_i to V_j when deleting a ϕ-function, would clearly be wasteful. Instead, a mapping table can be used, which would map V_i to V_j; all references to variable instances in ϕ-functions would be filtered through this table.

This technique is well-known, under several different names: the equivalence problem [12, 18], set merging [15], disjoint set union [25].

Improvement 3. *Basic blocks with single predecessors.*

Some ϕ-functions will always be deleted immediately. If a block has only a single predecessor, then it can't be a join point, so a ϕ-function need not be placed there during the RC phase.

At first sight, this improvement is incompatible with Improvement 1, which assumes ϕ-functions in every block. When combined with Improvement 2, however, this difficulty can be overcome. Upon finding a block with a single predecessor, the mapping table can simply be primed accordingly, instead of creating the ϕ-function.

4 Timing Results

We implemented our algorithm with the above improvements as a drop-in replacement for the Cytron et al. algorithm [10] used in the Gardens Point Modula-2 compiler. Timings were conducted on a 200 MHz Pentium with 64 M of RAM and a clock granularity of 10 ms, running Debian GNU/Linux version 2.1. To minimize transient timing errors, we ran each test five times; the times reported are the arithmetic mean of those five runs.

Figure 9 shows the time both algorithms take on a sample of thirty source files, comprising approximately 26,000 lines of code. (This code is the Modula-2 compiler's front end.) For all but a few of the files, our algorithm is competitive, sometimes faster than Cytron's.

What is often overlooked is the fact that SSA-generation algorithms do not operate in a vacuum. It is revealing to look at our algorithm in context. Figure 10 shows that, compared to the entire compilation, our algorithm takes an insignificant amount of time — this "total time" does not even include the time taken by the compiler's front end! Given that SSA generation time is not even a remotely dominant factor in compilation time, a simple algorithm such as ours may reasonably be used.

5 Related Work

In this section we survey other methods for converting to SSA form. These methods have been categorized based on the largest class of control-flow graph (CFG) they operate on: reducible or nonreducible.

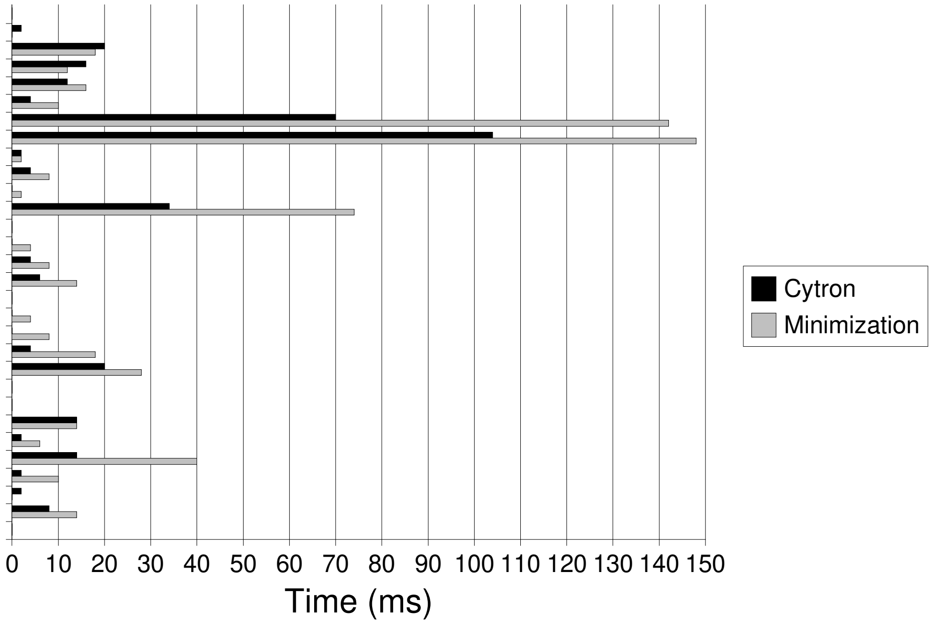


Fig. 9. Timing results.

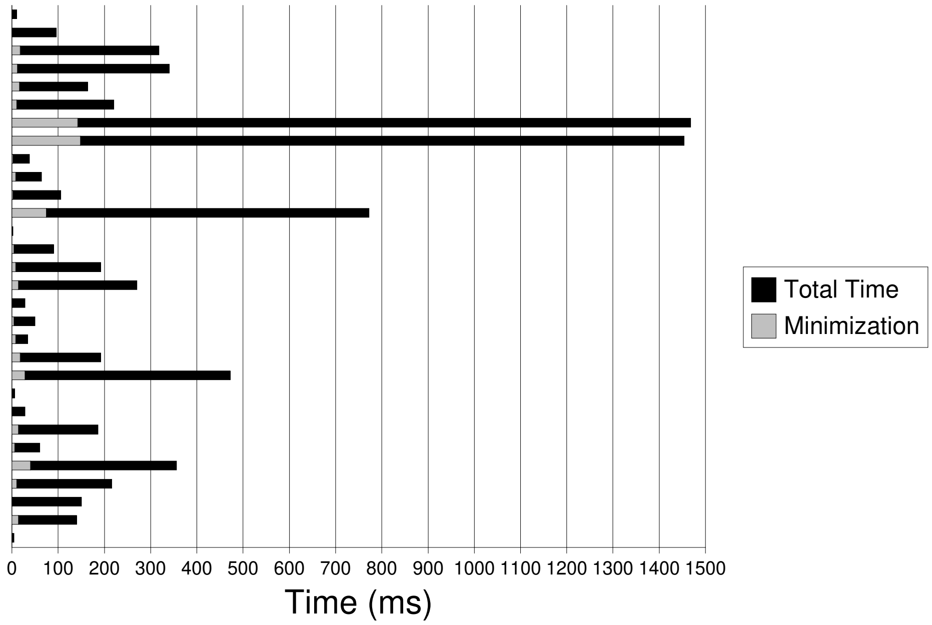


Fig. 10. Results in context.

5.1 Reducible CFGs

Brandis and Mössenböck [5] generate SSA form in one pass for structured control-flow graphs, a subset of reducible control-flow graphs, by delicate placement of ϕ-functions. They describe how to extend their method to reducible control-flow graphs, but require the dominator tree to do so.

Cytron, Lowry, and Zadeck [11] predate the use of ϕ-functions, and employ a heuristic placement policy based on the interval structure of the control-flow graph, similar to that of Rosen, Wegman, and Zadeck [22]. The latter work is interesting because they look for the same patterns as our algorithm does during our minimization phase. However, they do so after generating SSA form, and then only to correct 'second order effects' created during redundancy elimination.

5.2 Nonreducible CFGs

The work of Cytron et al. [10] is the method for generating SSA form we described in Sect. 2. Cytron and Ferrante [9] later refined their method so that it runs in almost-linear time.

Johnson, Pearson, and Pingali [16] demonstrate conversion to SSA form as an application of their "program structure tree," a decomposition of the control-flow graph into single-entry, single-exit regions. They claim that using this graph representation allows them to avoid areas in the control-flow graph that do not contribute to a solution.

The genesis of SSA form was in the 1960s with the work of Shapiro and Saint [23, 19]. Their conversion algorithm was based upon finding equivalence classes of variables by walking the control-flow graph.

Finally, Sreedhar and Gao [24] devised a linear-time algorithm for ϕ-function placement using DJ-graphs, a data structure which combines the dominator tree with information about where data flow in the program merges.

All of the algorithms for nonreducible control-flow graphs described in this subsection have been proven to yield a minimal placement of ϕ-functions.

6 Future Work

There are a number of avenues for further work. First, we would like to determine the time complexity of our algorithm, although this is unlikely to matter in practice — there is evidence suggesting that some of these algorithms only rarely achieve worst-case performance [9, 10].

Second, our algorithm may be extendible to other forms of minimal SSA, such as "pruned" SSA form, which only places a ϕ-function if the variable is live at that point [7].

Third, we are currently throwing away useful information. The algorithm of Cytron et al. that we compare our algorithm to in Sect. 4 only determines where ϕ-functions should be placed. Our algorithm determines this too, of course, but also knows upon completion what the arguments to the ϕ-functions are, something Cytron's algorithm does not know until variable renaming. It is possible that we can concoct a faster and/or simpler variable renaming algorithm as a result.

7 Applications

Our algorithm is particularly suitable in applications where a simple algorithm would be preferred, without the baggage of extra data structures. One might argue that the "extra" information computed by other algorithms will be used later: in fact, Cytron et al. suggest this [10]. However, in the two compilers we found employing SSA, neither made further use of the iterated dominance frontier information.

Some optimizations necessitate the re-generation of minimal SSA form. For example, re-minimization is required in the SSA-based partial redundancy elimination algorithm of [8], for which they suggest re-running the ϕ-insertion algorithm. Other optimizations may force SSA re-generation by changing the structure of the control-flow graph. Our algorithm may be useful in these situations.

8 Conclusions

We have presented a new, simple method of generating SSA form which finds a minimal ϕ-function placement for an important class of control-flow graph — reducible ones — and which finds a correct placement for all control-flow graphs, even nonreducible ones. Our timings indicate that it is competitive with the prevalent method of generating SSA form, especially when viewed in context.

9 Acknowledgments

Many thanks to John Gough for use of the Gardens Point Modula-2 compiler. The IFIP Working Group 2.4 made a number of helpful comments; in particular, Bob Morgan suggested applying our algorithm for re-minimization. The anonymous referees made many helpful comments. This work was supported in part by a grant from the National Science and Engineering Research Council of Canada.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. Code Optimization and Finite Church-Rosser Systems. In *Design and Optimization of Compilers*, R. Rustin, ed. Prentice Hall, 1971, pp. 89–105.
2. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988, pp. 1–11.
3. A. W. Appel. SSA is Functional Programming. *ACM SIGPLAN 33*, 4 (April 1998), pp. 17–20.

4. A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge, 1998.
5. M. M. Brandis and H. Mössenböck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM TOPLAS 16*, 6 (November 1994), pp. 1684–1698.
6. P. Briggs, T. Harvey, and T. Simpson. Static Single Assignment Construction, Version 1.0. Unpublished document, 1995.
7. J.-D. Choi, R. Cytron, and J. Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. *ACM POPL '91*, pp. 55–66.
8. F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A New Algorithm for Partial Redundancy Elimination based on SSA Form. *ACM PLDI '97*, pp. 273–286.
9. R. K. Cytron and J. Ferrante. Efficiently Computing ϕ-Nodes On-The-Fly. *ACM TOPLAS 17*, 3 (May 1995), pp. 487–506.
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 TOPLAS 13*, 4 (October 1991), pp. 451–490.
11. R. Cytron, A. Lowry, K. Zadeck. Code Motion of Control Structures in High-Level Languages. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986, pp. 70–85.
12. M. J. Fischer. Efficiency of Equivalence Algorithms. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds. Plenum Press, 1972.
13. M. S. Hecht. *Flow Analysis of Computer Programs*, North-Holland, 1977.
14. M. S. Hecht and J. D. Ullman. Flow Graph Reducibility. *SIAM Journal of Computing 1*, 2 (June 1972), pp. 188–202.
15. J. E. Hopcroft and J. D. Ullman. Set Merging Algorithms. *SIAM Journal of Computing 2*, 4 (December 1973), pp. 294–303.
16. R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. *ACM PLDI '94*, pp. 171–185.
17. D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software — Practice and Experience 1*, 1971, pp. 105–133.
18. D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley, 1997.
19. D. B. Loveman and R. A. Faneuf. Program Optimization — Theory and Practice. *Conference on Programming Languages and Compilers for Parallel and Vector Machines*, 1975, pp. 97–102.
20. R. Morgan. *Building an Optimizing Compiler*, Digital Press, 1998.
21. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
22. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988, pp. 12–27.
23. R. M. Shapiro and H. Saint. The Representation of Algorithms. Rome Air Development Center TR-69-313, Volume II, September 1969.
24. V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing ϕ-Nodes. *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, 1995, pp. 62–73.
25. R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *JACM 22*, 2 (April 1975), pp. 215–225.
26. J. D. Ullman. Fast Algorithms for the Elimination of Common Subexpressions. *Acta Informatica 2*, 1973, pp. 191–213.
27. M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM TOPLAS 13*, 2 (April 1991), pp. 181–210.