Single-Pass Generation of Static Single Assignment Form for Structured Languages

MARC M. BRANDIS and HANSPETER MÖSSENBÖCK
ETH Zürich, Institute for Computer Systems

Over the last few years, static single assignment (SSA) form has been established as a suitable intermediate program representation that allows new powerful optimizations and simplifies others considerably. Algorithms are known that generate SSA form from programs with an arbitrary flow of control. These algorithms need several passes. We show that it is possible to generate SSA form in a single pass (even during parsing) if the program contains only structured control flow (i.e., no gotos). For such programs the dominator tree can be built on the fly, too.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, optimization*. D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*. E.1 [**Data Structures**]: Trees.

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Dominator tree, static single assignment form, structured languages

---

1. INTRODUCTION

Over the last few years, *static single assignment (SSA) form* [5] has been established as a suitable intermediate program representation that allows new powerful optimizations and simplifies others considerably. The essential property of SSA form is that there is only one assignment to each variable in the whole program text. This makes it easy to reason about variables: If two variables have the same name they also contain the same value.

1.1 Informal explanation of SSA form

We first explain the construction of SSA form informally before we turn to the description of our algorithm. Let's start with a simple sequence of assignments. To obtain its SSA form, we transform it so that in every assignment the variable on the left-hand side is given a unique name and all uses of this variable are renamed accordingly (Figure 1). We use subscripts to make variable names unique and call the subscripted variables *value instances* of the original variables, or *values* for short.

This research was supported in part by the Swiss National Science Foundation (Schweizerischer Nationalfonds zur Förderung der wissenschaftlichen Forschung) under grant 21-33459.92 and in part by ETH Zürich.
Authors´ addresses: M. M. Brandis, ETH Zurich, Institute for Computer Systems, CH-8092 Zurich, Switzerland; H. Mössenböck, currently at University of Linz, Institute for Computer Science, A-4040 Linz, Austria.

```
v := 0;                    v1 := 0;
x := v + 1;                x1 := v1 + 1;
v := 2;                    v2 := 2;
y := v + 3                 y1 := v2 + 3
```

Fig. 1. Assignment sequence in original form and in SSA form

More complicated programs contain branches and *join nodes*. At join nodes, multiple values of a variable may reach the node via different branches. These values have to be merged into one single value that reaches further uses of this variable and for which there is again one single assignment (see Figure 2). For this purpose assignments are generated with so-called ϕ-functions on the right-hand side. ϕ-functions have as many operands as there are branches into the join node. Thus the ϕ-function in Figure 2 has two operands. The meaning of a ϕ-function is: if control reaches the join node via the *i*-th branch the value of the function is its *i*-th operand.

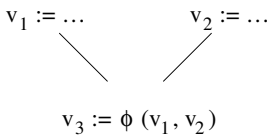

Fig. 2. Values v_1 and v_2 have to be merged into a unique value v_3

Figure 3 shows the *control flow graphs* [1] for an IF statement and a WHILE statement with instructions in SSA form. The nodes of the graphs are *basic blocks*, i.e., instruction sequences with a single entry and no branch instruction, except possibly at the end of the sequence.

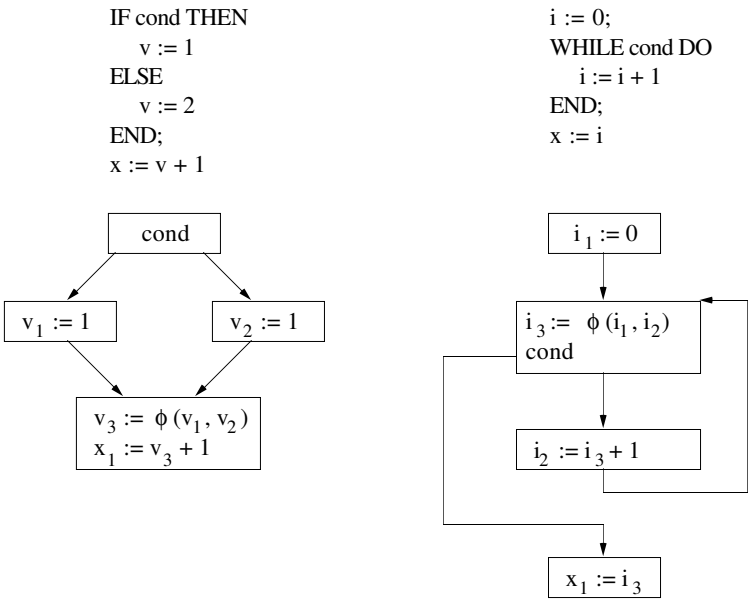

Fig. 3. Control flow graphs of an IF and a WHILE statement with instructions in SSA form

The single-assignment property simplifies reasoning about variables, since for every value instance its (single) defining assignment is known. Because every assignment creates a new value name it cannot *kill* (i.e., invalidate) expressions previously computed from other values. In particular, if two expressions are textually the same, they are sure to evaluate to the same result.

Efficient and powerful optimization algorithms based on SSA form have been described for constant propagation [12], common subexpression elimination [2], partial redundancy elimination [11], code motion [4], and induction variable analysis [15]. The single-assignment property is also helpful in instruction scheduling as it leaves only the essential data dependences in the code and avoids output- and anti-dependences [10]. SSA form has also been adapted for table-driven code generation in [7].

1.2 Algorithms for generating SSA form

Cytron et al. [5] have presented an efficient algorithm for generating SSA form from the instructions of an arbitrary control flow graph and its *dominator tree* [1]. Their algorithm computes the *dominance frontiers* in the graph, which are the nodes where ϕ-functions have to be placed. This is the most efficient algorithm currently known for a general flow graph but it requires several passes over the instructions of the graph.

Structured control flow graphs simplify the generation of SSA form. Although this seems to be kind of "folklore" knowledge it has not been published in detail to the best of our knowledge. Rosen et al. [11] sketch a single-pass algorithm for generating SSA form but their algorithm requires a topologically sorted control flow graph and generates many unnecessary ϕ-assignments. A similar idea was exploited by Horwitz et al. [8] in the construction of dependence graphs.

In a recent paper [9] Johnson and Pingali describe a method for computing the dependence flow graph of a program and deriving SSA form from it. They identify single-entry/single-exit regions to place *merge nodes* which are similar to ϕ-assignments. Although their method is similar to ours in that it makes use of structured control flow it is inherently multi-pass.

In this paper we present a technique for generating SSA form in a single pass directly from the source text of a program. It can be applied to structured programs, i.e., to programs that contain only assignments and structured statements (such as IF, CASE, WHILE, REPEAT, or FOR) but no goto statements. For such programs the join nodes and thus the places where to insert ϕ-assignments are immediately known so that ϕ-assignments can be generated on the fly during parsing. This is useful for languages such as Simula[3], Modula-2 [13] or Oberon [14] that lack goto statements at all. If language independence is desired our algorithm can also be applied to control flow graphs instead of source programs.

The advantage of generating SSA form directly during parsing is that it saves an intermediate step. Instead of building a high-level representation of a program and then transforming it, we directly generate machine-specific instructions in SSA form ready for optimizations. This saves time and memory. However, the same technique could be used for generating a machine-independent representation as well.

We do not deal here with alias problems caused by assignments to array elements, to parameters

passed by reference, and to variables that are referenced via pointers. These problems can be dealt with in the same way as described in [5].

While Cytron's method requires the construction of the dominator tree—itself a non-trivial step—our method does not need such a data structure. However, the dominator tree is useful in subsequent optimizations so that it is worth showing that for structured programs it can be built in a single pass during parsing, too.

Section 2 of this paper explains our algorithm for generating SSA form while Section 3 shows how to build the dominator tree during parsing. Section 4 demonstrates how to extend our method to certain unstructured language features. Section 5 estimates the time bounds of our algorithm, Section 6 shows measurements, and Section 7 draws some conclusions.

2. COMPUTING STATIC SINGLE ASSIGNMENT FORM

2.1 Naming of values

Every assignment to a variable v generates a new value v_i where i is a unique number for this variable. After the assignment, v_i is the *current value* of v. Every subsequent use of v is replaced by a use of its current value (see Figure 4). The current value of a variable can be stored in its symbol table entry.

Assignments (original form)	*Assignments (SSA form)*	*Current values* v	x
		v_0	x_0
v := 0;	v_1 := 0;	v_1	x_0
x := v + 1;	x_1 := v_1 + 1;	v_1	x_1
v := 2	v_2 := 2	v_2	x_1

Fig. 4. Current values of v and x during the compilation of an assignment sequence

2.2 Join nodes

Control structures such as IF and WHILE statements introduce branches into the flow graph of a procedure. A node where two branches join is called a *join node*. For all structured statements the corresponding join nodes are known in advance (see Figure 5). To the flow graph of every procedure we add a virtual start node *Enter* and a virtual end node *Exit*. We introduce an empty branch from Enter to Exit in order to make Exit a join node.

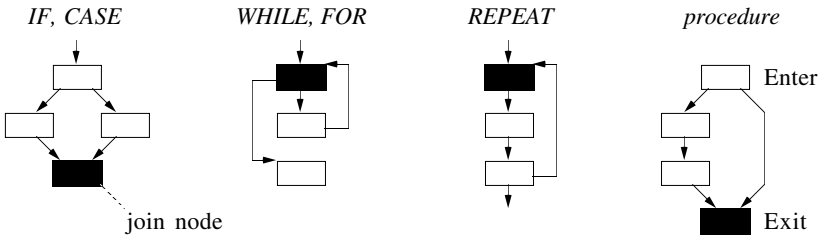

Fig. 5. Control structures and their join nodes in the flow graph

Control structures may be nested. The join node of the innermost control structure currently being compiled is called the *current join node*.

Note that we do not introduce special nodes for joins but reuse the nodes which naturally appear at the points where branches are merged. For example, the join node of an IF statement will also contain the statements following the IF. The only case where we introduce additional join nodes is when REPEAT statements are nested (Figure 6).

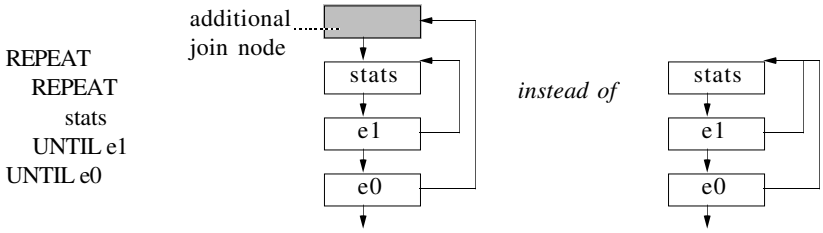

Fig. 6. Additional join nodes may be inserted for nested REPEAT statements

This is necessary for later optimizations. For example, if the join nodes of the two loops were merged one could not move loop invariant calculations out of the innermost loop.

2.3 *Where to place ϕ-assignments*

Every assignment belongs to a certain branch of the enclosing control structure. It introduces a new value and when this value reaches the next join node it will be different from the value(s) of the same variable that reach the join node via other branches. Therefore, every assignment to a variable v adds or modifies a ϕ-assignment for v in the current join node. The ϕ-operand corresponding to the branch that contains the assignment is replaced by the current value of v (see Figure 7).

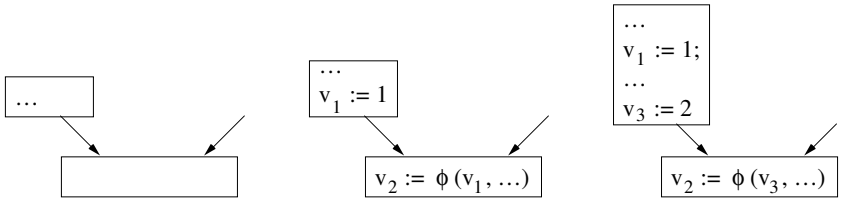

Fig. 7. Insertion (modification) of a ϕ-assignment when compiling assignments

Note that ϕ-assignments, like all assignments, generate new values and therefore cause the placement of other ϕ-assignments in the next outer join node (with the exception of ϕ-assignments in the Exit node for which there is no join node).

2.4 Compiling IF statements

When we start parsing an IF statement we immediately create a new join node which will later be linked to the control flow graph. This join node becomes the container of all ϕ-functions generated due to assignments in both branches of the IF statement.

An assignment to a variable v in the THEN part will create a new current value v_j which is different from the current value v_i at the beginning of the ELSE part (see Figure 8). Therefore, the old current value v_i has to be remembered as a *backup value* and has to be restored before the ELSE part is parsed. We conveniently store the backup value of v in the ϕ-function created by the first assignment to v in a statement sequence. At the end of the statement sequence we traverse the ϕ-functions in the current join node and reset the current values to the stored backup values. Thus every statement sequence leaves the current values unchanged.

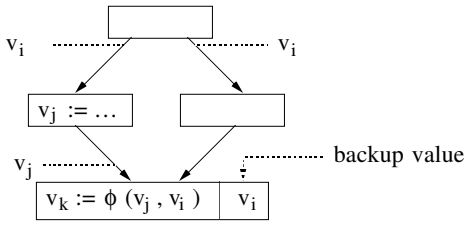

Fig. 8. Current values of v in both branches of an IF statement. v_i is stored as a backup value in the ϕ-function of the join node.

Figure 9 shows various snapshots of the control flow graph during the translation of an IF statement. After snapshot 3 the current value of a is reset to a_0 and the current value of b is reset to b_0. Note that there is no assignment to b in the ELSE branch and no assignment to c in the THEN branch. The corresponding operands in the ϕ-functions for b and c are therefore the current values of b and c at the beginning of the branches, namely b_0 and c_0 (the backup values).

When the whole IF statement has been parsed the generated ϕ-assignments are *committed*, i.e., they are processed as assignments. This causes the placement of other ϕ-assignments in the join node of the enclosing control structure and the values on the left-hand sides of the ϕ-assignments (a_2, b_2, c_2) become the new current values for a, b, and c.

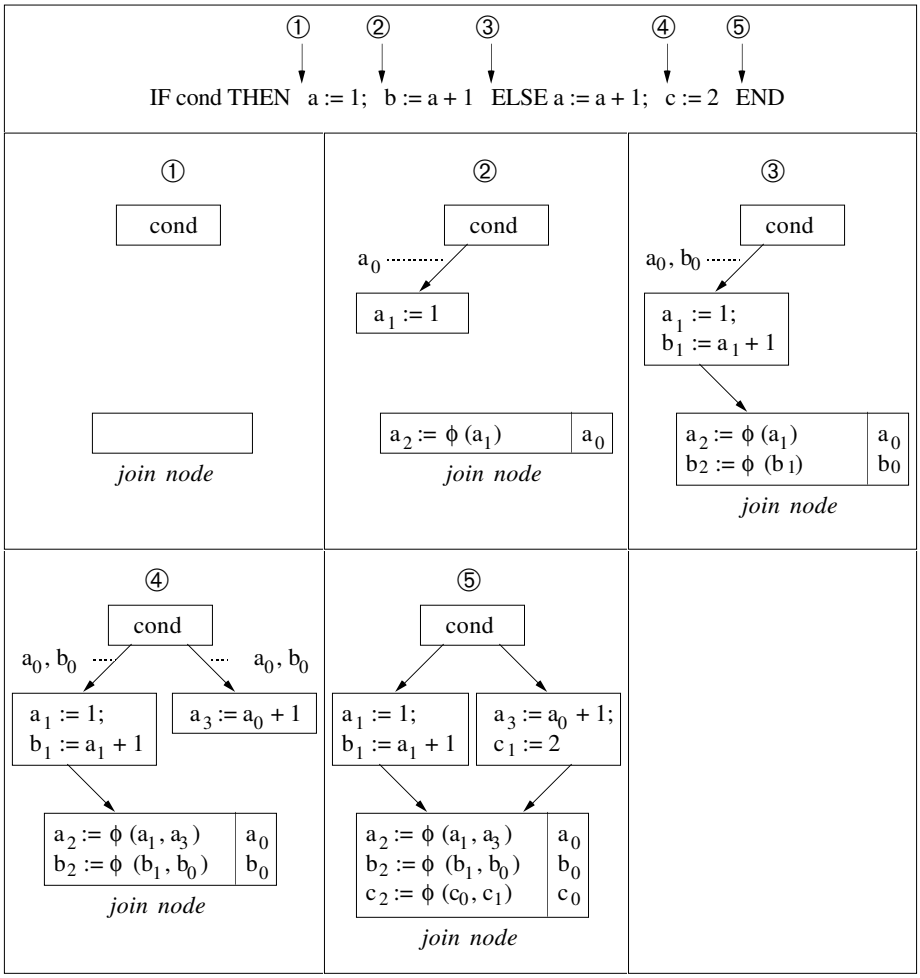

Fig. 9. Generation of SSA form for an IF statement

2.5 Compiling WHILE statements

The join node of a WHILE statement is the *header node* [1] of the loop, i.e., the node where the entry to the loop and the backward branch join. All assignments in the loop cause the placement of ϕ-functions in this join node in the same way as it is done for IF statements.

When a ϕ-function is inserted in the join node of a WHILE statement this leads to a new current value for a variable that might have already been used in the loop. All uses of this variable must thus be replaced by the new current value. For every value we keep a list of instructions where this value is used (a *use chain*). By traversing the use chain of a value through all instructions whose address is larger than the address of the loop header it is easy to replace all occurrences of the value in the loop. Figure 10 shows some snapshots of the control flow graph during the translation of a WHILE statement.

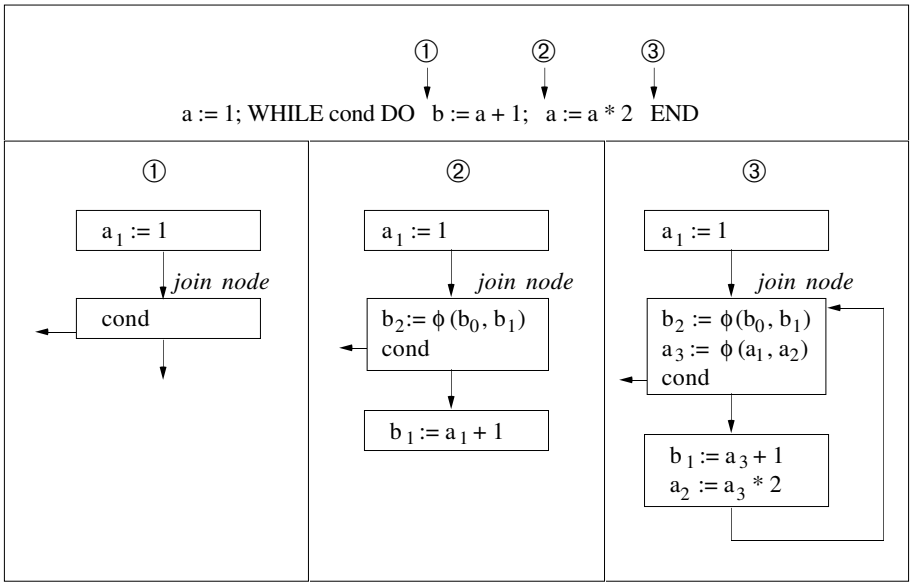

Fig. 10. Generation of SSA form for a WHILE statement. Note that a_1 has been replaced by a_3 in ③.

When the whole WHILE statement has been parsed the ϕ-assignments in its join node are committed. This causes the placement of new ϕ-functions in the next outer join node and makes b_2 and a_3 the new current values of b and a for the instructions following the WHILE statement. (Note: if the language allows the condition in the WHILE header to contain assignments, the values created by these assignment have to be taken as the current values after the WHILE statement.)

The compilation of CASE statements and FOR statements follows the same lines as the compilation of IF statements and WHILE statements.

2.6 Compiling REPEAT statements

REPEAT statements are special because control does not leave them via their join node. The join node is the loop header and all assignments in the REPEAT statement cause the placement of ϕ-functions in it (as for WHILE statements). However, when these ϕ-assignments are committed after the REPEAT statement, the second operand of every ϕ-function (the one corresponding to the backward branch of the loop) is taken as the new current value after the REPEAT statement. This is also the value that becomes an operand in the ϕ-function of the next outer join node (see Figure 11).

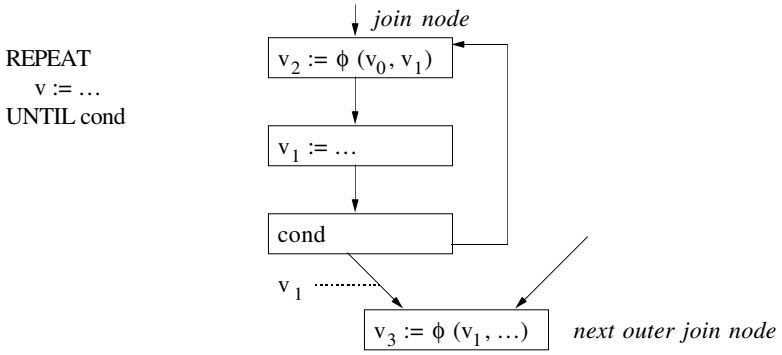

Fig. 11: Generation of SSA form for a REPEAT statement. $v1$ is the current value of v after the REPEAT statement.

2.7 Implementation

SSA form can be generated using a procedure *InsertPhi* to generate or modify a ϕ-assignment and a procedure *CommitPhi* to commit the ϕ-assignments of a join node.

Whenever an assignment "v_i := …" is encountered in the i-th branch leading to join node b, *InsertPhi*(b, i, v_i, v_{old}) is called, where v_{old} is the value of v before the assignment. This value is stored as the backup value in the ϕ-assignment (denoted as "v_j := ϕ (…) / v_{old}").

```
PROCEDURE InsertPhi (b: Node; i: INTEGER; vi, vold: Value);
BEGIN
   IF b contains no ϕ-assignment for v THEN
      Insert "vj := ϕ (vold, …, vold) / vold" in b;
      IF b is a join node of a loop THEN
         Rename all mentions of vold in the loop to vj
      END
   END;
   Replace i-th operand of v's ϕ-assignment by vi
END InsertPhi;
```

Whenever a join node b contained in the I-th branch of its outer join node B is committed, *CommitPhi*(b) is called.

```
PROCEDURE CommitPhi (b: Node);
BEGIN
   FOR all ϕ-instructions "vi := ϕ (v0, …, vn) / vold" in b DO
      IF b is a join node of a repeat THEN val := vn ELSE val := vi END;
      Make val the current value of v;
      InsertPhi(B, I, val, vold)
   END
END CommitPhi;
```

3. COMPUTING THE DOMINATOR TREE

Our method for generating SSA form does not need the dominator tree. However, many optimization algorithms require this data structure, and we will show another use in Section 4. Therefore, we show that for structured programs the dominator tree can again be built on the fly during parsing.

The dominator tree is based on the *dominance relation* [1] between basic blocks: A block X is said to *dominate* a block Y if X appears on every path from the start node of the flow graph to Y. X is called a *dominator* of Y. If X is the closest dominator of Y on any path from the start node to Y then X is called the *immediate dominator* of Y. A *dominator tree* is a tree in which the father of every block is its immediate dominator.

An efficient algorithm to construct the dominator tree for arbitrary flow graphs has been described in [6], but it requires several passes over the graph. For structured programs we can use a much simpler technique which allows us to build the dominator tree in a single pass during parsing.

Structured statements such as IF, WHILE, FOR, REPEAT, and CASE statements are subgraphs with a single entry node and a single exit node. The entry node dominates all nodes inside the statement and the immediate dominator of inner nodes is defined as shown in Figure 12. Thus we can set the immediate dominator of every node immediately when we create the node.

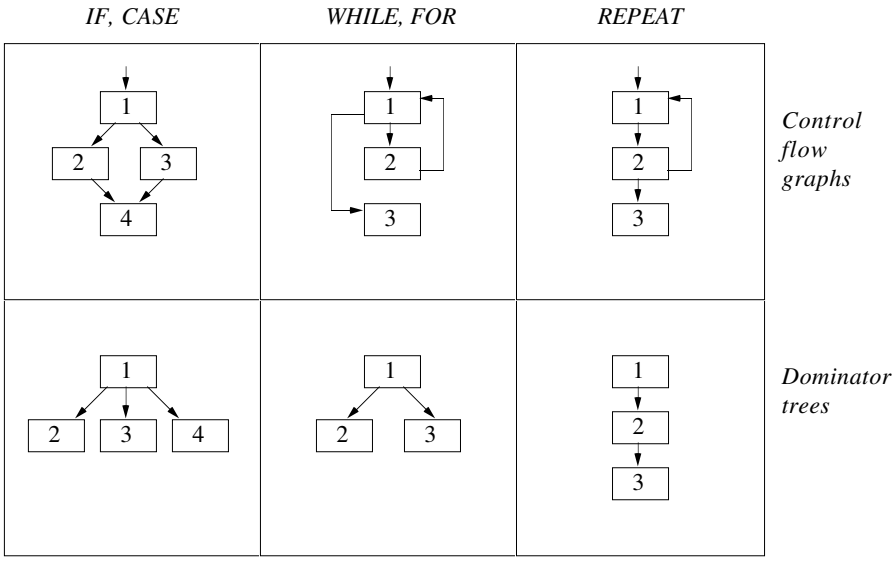

Fig. 12. Structured control flow graphs and their dominator trees

4. EXTENDING OUR METHOD TO UNSTRUCTURED STATEMENTS

For structured statements we have shown how to generate both SSA form and the dominator tree in a single pass during parsing. In the following section we will show that it is even possible to extend our method to a certain class of unstructured statements (LOOP/EXIT and RETURN) that may cause exits from control structures at arbitrary points. However, since such exits are a kind of (disciplined) goto it

is not surprising that they are much harder to handle than structured statements.

We do not elaborate on the processing of unstructured statements here but simply want to show that such statements can be handled in a single pass, too.

4.1 Compiling LOOP and EXIT statements

A LOOP statement (see for example [14]) is an endless loop that can be left at various points by an EXIT statement. It introduces two join nodes: one is the loop header where the entry to the loop and the backward branch join; the other is the node following the LOOP statement which is the target of all exit branches (see Figure 13). All ϕ-functions generated due to assignments directly in the loop are collected in the join node at the loop header. However, after the LOOP statement the ϕ-assignments in the loop header are not committed, since these are not the assignments which determine the new current values after the LOOP statement. Rather, the dominator tree is traversed from every node containing an EXIT statement to the loop header and all assignments on this path cause the placement of ϕ-functions in the join node after the LOOP statement. These ϕ-assignments are finally committed.

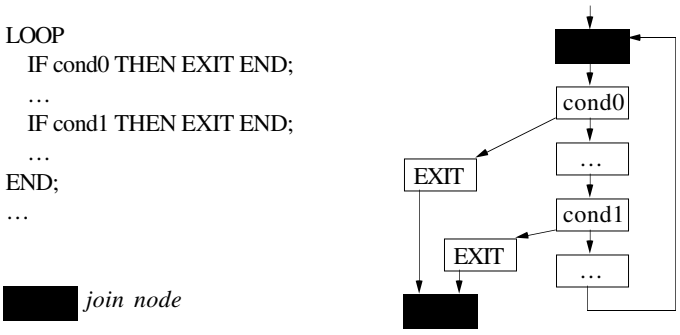

Fig. 13. Join nodes in a LOOP statement with several exit points

Every EXIT statement removes an incoming branch from its current join node. Therefore the ϕ-operands corresponding to this branch must be removed. ϕ-functions which end up having no operands at all are deleted.

For programs containing EXIT statements the construction of the dominator tree becomes more complicated, too. An EXIT statement removes a branch from the current join node thus possibly changing the immediate dominator of this node. Consider Figure 14: The dominator tree of the control flow graph (a) is shown in (b). However, when node 4 contains an EXIT statement, the dominator tree changes to (c). When both node 3 and node 4 contain an EXIT statement node 5 is not reachable any more and the dominator tree changes to (d).

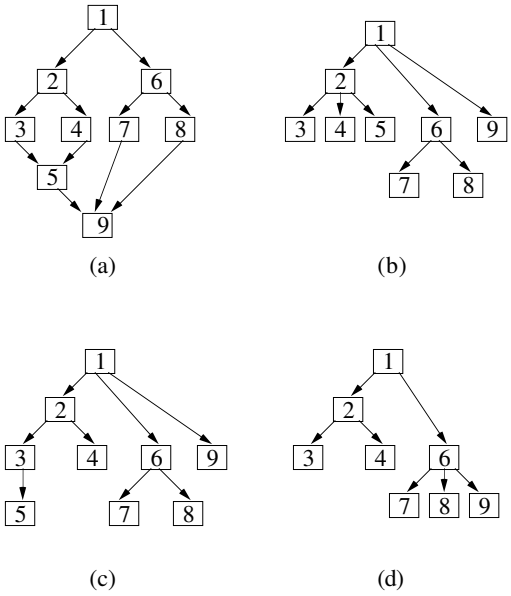

Fig. 14. A control flow graph (a), its dominator tree without EXITs (b), with an EXIT in node 4 (c) and with EXITs in nodes 3 and 4 (d)

When building the dominator tree one simply has to remember whether the statement sequence of a branch directly contains an EXIT statement. In this case the branch is said to be *dead*. The immediate dominator of a join node has to be computed as the *closest common dominator* on all incoming branches that are not dead. In order to find the closest common dominator of two nodes X and Y we traverse the dominator tree upwards from X and Y and take the first node which is on both paths.

Let us look at Figure 14 again. If node 4 in Figure 14 a contains an EXIT statement the second branch leading into node 5 is dead, therefore the immediate dominator of 5 is 3 (Figure 14 c). If both 3 and 4 contain an EXIT statement all branches leading into 5 are dead thus 5 does not have a dominator at all. Furthermore the first branch leading into 9 is dead, therefore the immediate dominator of 9 is the closest common dominator of 7 and 8 (the predecessors on the branches that are still alive) which is 6 (Figure 14 d).

For the node following a LOOP statement the immediate dominator is also computed as the closest common dominator of all incoming branches, i.e., of all nodes containing an EXIT statement that leads to this target node.

4.2 Compiling RETURN statements

RETURN statements (used to terminate a procedure and to return control to its caller) can be handled in the same way as EXIT statements. A RETURN statement makes a branch dead and thus removes it from the current join node. As for EXIT statements, the corresponding operands of all ϕ-functions in this join node have to be removed. The immediate dominator of the join node has to be computed as the closest common dominator of all live branches into this join node.

5. ANALYSIS

Cytron et al. [5] report that the run time complexity of their algorithm is $O(r^3)$ in the worst case, where r is the maximum of either the number of nodes in the control flow graph, the number of edges in this graph, the number of original assignments, and the number of original mentions of a variable in the program. We found that our algorithm has the same worst-case complexity. This is not surprising since the size of the SSA form, i.e., the number of ϕ-assignments and the number of their operands, is independent of the algorithm used for its generation. In practice, however, the translation to SSA form is $O(r)$ for both algorithms. We found that every original assignment generates 1.2 ϕ-assignments on the average. These measurements are in correspondence with the numbers reported in [5].

The run time of our algorithm is determined by the effort to generate the ϕ-assignments and to rename the mentions of variables in a loop after ϕ-assignments for this variables have been inserted in the loop header (see Section 2.5). Every original assignment generates at most one ϕ-assignment in every join node. If a is the number of original assignments and b is the number of join nodes (one for every statement like IF, WHILE, etc.), the number of ϕ-assignment is $O(ab)$. The mentions that have to be renamed in a loop are at most the original mentions in the loop (m) plus the mentions in ϕ-assignments contained in the loop (p). Since a ϕ-assignment has at most n mentions (if n is the maximum number of branches leading into a join node), p is $O(abn)$. The total run time is therefore $O(ab) + O(m) + O(abn)$. If r is the maximum of a, b, m and n then the run time is $O(r^3)$ in the worst case.

Although our algorithm has the same worst-case behavior as the one described in [5], it generates SSA form in a single pass while the algorithm in [5] needs three passes plus additional effort to build the dominator tree. Our algorithm looks at every assignment exactly once, immediately generating a ϕ-operand in the right join node. Furthermore we usually choose the appropriate name for every mention of a variable immediately when generating the instructions, while the algorithm in [5] needs a separate pass to rename all mentions after the ϕ-assignments have been placed. This makes us believe that our algorithm runs faster and is simpler to implement. However, this has still to be proved by empirical data and is a topic of future research.

6. MEASUREMENTS

We implemented our algorithm in a compiler for a subset of Oberon (with expressions, assignments, IF, WHILE, REPEAT, and LOOP/EXIT statements, as well as scalar variables and arrays). The generation of SSA form and the construction of the dominator tree made up 250 lines of source code which accounted for 15% of the compiler's total source code.

Table I shows some data from the compilation of three benchmark programs: Sieve (calculating prime numbers), Heapsort, and Polyphase sort. We measured the total number of generated instructions including ϕ-assignments (I_ϕ) and the number of generated instructions without ϕ-assignments (I). Furthermore we measured the time to generate the instructions from the source code with generation of ϕ-assignments (T_ϕ) and without generation of ϕ-assignments (T). The measurements

were performed on a NS32532 processor running at 25 MHz.

Table I

	Statements	Generated instructions			Compilation time (in ms)		
		I_ϕ	I	%	T_ϕ	T	%
Sieve	17	48	42	13	33	30	9
Heapsort	46	162	138	15	90	83	8
Polyphase sort	104	514	426	17	243	220	9

The results show that our algorithm imposes a linear time penalty of about 10% and an almost linear space overhead of about 15% to a compilation in practice.

7. CONCLUSIONS

Static single assignment form and the dominator tree are important data structures in optimizing compilers but they are difficult to generate for programs with an arbitrary flow of control. For structured programs without goto statements it is possible to build both SSA form and the dominator tree in a single pass directly during parsing. We have shown a straight-forward technique for that.

It is remarkable how much the omission of the goto statement simplifys the algorithms to generate SSA form. It demonstrates how a poor language feature may cause undue overhead in a compiler. We regard it as an important result that the restriction of a language to a few simple control structures also leads to a simpler and more efficient optimizing compiler.

Our algorithms do not only allow building small and efficient compilers, but they are also easy to understand. It was possible to teach them in an undergraduate course within a few weeks (2 hours per week) while the students built a full optimizing compiler for a small language based on SSA form. Given the ease with which the powerful and elegant SSA form can be generated for structured languages, we do not see any reason why compilers for such languages should build any other intermediate program representation such as an abstract syntax tree.

We implemented our algorithms in a compiler for a subset of Oberon. Both the generation of SSA form and the construction of the dominator tree could be implemented with less than 250 lines of code. We currently work on integrating our technique in a full Oberon compiler.

ACKNOWLEDGEMENTS

We would like to thank Robert Griesemer and Josef Templ for their valuable comments on earlier versions of this paper as well as the anonymous referees for their many helpful suggestions.

REFERENCES

1. AHO, A.V., SETHI, R., AND ULLMAN, J.D. *Compilers*. Addison-Wesley, 1986.
2. ALPERN, B., WEGMAN, M.N., AND ZADECK, F.K. Detecting equality of variables in programs. *In Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (1988), 1-11.
3. BIRTWISTLE, G.M., DAHL, O.-J., MYHRHAUG, B., NYGAARD, K.: *Simula Begin*. Studentlitteratur, Lund, Sweden, 1979.
4. CYTRON, R., LOWRY, A., AND ZADECK, F.K. Code motion of control structures in high-level languages. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (1986), 70-85.
5. CYTRON, R., FERRANTE, J., ROSEN, B.K., WEGMAN, M.N., AND ZADECK, F.K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451-490.
6. LENGAUER, T., AND TARJAN, R.E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121-141.
7. MCCONNELL, C., AND JOHNSON, R.E. Using static single assignment form in a code optimizer. *ACM Letters on Programming Languages and Systems*, 1, 2 (1992), 152-160.
8. HORWITZ, S., PRINS, J., AND REPS, T. Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst*. 11, 3 (1989), 345-387.
9. JOHNSON, R., PINGALI, K. Dependence-based program analysis. In *Proceedings of the SIGPLAN'93 Symposium on Programming Languages Design and Implementation*(1993), 78-89.
10. NICOLAN, A., POTASMAN, R., AND WANG, H. Register allocation, renaming, and their impact on fine-grain parallelism. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, Springer-Verlag (1991), 218-235.
11. ROSEN, B.K., WEGMAN, M.N., AND ZADECK, F.K. Global value numbers and redundant computations. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (1988), ACM, New York, 12-27.
12. WEGMAN, M.N., AND ZADECK, F.K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst*. 13, 2, (1991), 181-210.
13. WIRTH, N. *Programming in Modula-2*. Springer-Verlag, 1982.
14. WIRTH, N., AND REISER, M. *Programming in Oberon, steps beyond Pascal and Modula-2*. Addison-Wesley, 1992.
15. WOLFE, M.. Beyond induction variables. In *Proceedings of the SIGPLAN'92 Symposium on Programming Languages Design and Implementation*, SIGPLAN Notices (1992), 162-174.