Practical Improvements to the Construction and Destruction of Static Single Assignment Form

Preston Briggs
Tera Computer Company

Keith D. Cooper
Timothy J. Harvey
L. Taylor Simpson
Rice University

Abstract

Static single assignment (SSA) form is a program representation that is useful for compiler-based code optimization. In this paper, we address three problems that have arisen in our use of SSA form. Two are improvements to the SSA construction algorithms presented by Cytron *et al.* [10]. The first improvement is a version of SSA form that we call "semi-pruned" SSA. The semi-pruned construction builds a smaller SSA form than the "minimal" form at a modest increase in cost. It avoids the global data-flow analysis required to build "pruned" SSA. The second improvement speeds up the program renaming process by efficiently manipulating the stacks of names. The stacks indicate the SSA name of each variable that reach a particular point in the program. Our improvement reduces the number of pushes performed in addition to more efficiently locating the stacks which should be popped. The final problem that we address concerns the process of converting SSA form back into executable code. The naive algorithm that appears to be prior practice can produce incorrect code in cases that involve either copy folding or "critical edges". This issue has caused earlier authors to restrict their work to SSA graphs without critical edges. We present an algorithm for replacing ϕ-nodes with copy instructions that generates correct code in the presence of both copy folding and critical edges.

We have implemented all of these improvements in our experimental optimizing compiler. For the two improvements to the SSA construction algorithm, we present experimental results that demonstrate the effectiveness of these improvements not only during the construction of SSA form, but also in the time saved by subsequent optimization passes that use a smaller representation of the program. For the copy insertion algorithm, correctness rather than performance is the issue. Our algorithm is slower than the naive algorithm, but it produces correct code. This should allow other authors to remove restrictions that they have placed on the shape of control-flow graphs used as input to their algorithms.

1 Introduction

Static single assignment (SSA) form is an intermediate representation that compilers use to facilitate program analysis and optimzation [9, 10]. SSA form can be viewed as a sparse representation for the information contained in classic use-definition and definition-use chains [16]. In many applications, it has become the primary program representation. SSA form has two principal advantages over prior representations.

1. It imposes a strict discipline on the name space used to represent values in the computation. Each reference of a name corresponds to the value produced at precisely one definition point.

2. It identifies the points in the computation where values from different control-flow paths merge. At a merge point, several different SSA names that correspond to different definitions of the same original name can flow together. To ensure the single-assignment property, the construction inserts a new definition at the merge point; its right hand side is a pseudo-function called a ϕ-function that represents the merge of multiple SSA names.

These properties simplify not only the building of data-flow analyses such as def-use and use-def chains, but they serve as a powerful framework in the analysis and design of optimization algorithms.

Converting code into SSA form can improve the results of analysis and simplify the implementation of various transformations. For example, the SSA name space contains a unique name for each definition point

Address all correspondence to Tim Harvey, Rice University, 6100 Main Street, Mail Stop 132, Houston, TX 77005, harv@rice.edu

in the procedure, both definition points that occurred in the original code and those inserted to embody the merging of different values from distinct control-flow paths. This leads to a larger name space; performing data-flow analysis over this new name space can provide the compiler with more detailed knowledge about the possible run-time flow of values. In transforming the code, the inserted ϕ-functions demarcate an important point in the flow of values. For example, any variable that is modified inside a loop will have a ϕ-function in the loop's header block. This feature makes identifying induction variables in loops particularly easy [21]. Since the introduction of SSA form in 1989, many papers have described applications of SSA form for analysis or transformation [2, 4, 20, 6].

The original work on SSA form presented two algorithms for constructing SSA form from the code for a procedure [9, 10]. The SSA constructed by the algorithms differs in the size of its name space and the number of ϕ-functions that must be inserted.

1. The "minimal" construction produces a form dubbed "minimal" SSA. It inserts a ϕ-function and definition at every point where a control-flow merge brings together two SSA names for a single original name. It can insert a ϕ-function to merge two values that are never used after the merge–in data-flow analysis terminology, two values that are not *live*.

2. The "pruned" construction produces a form dubbed "pruned" SSA. The pruned construction uses global data-flow analysis to decide where values are live. It only inserts ϕ-functions at those merge points where the analysis indicates that the value may be live. This can drastically reduce the number of ϕ-functions and, thus, the number of SSA names.

The two algorithms differ in their time and space complexity. The minimal algorithm avoids computing live information, so it is less expensive than the pruned algorithm. The consequence of algorithmic speed is a larger SSA form.

This paper presents three algorithmic improvements to the current art of building and using SSA form. The first creates a SSA form that has fewer nodes than the minimal form without the expense of solving data-flow equations to determine which values are "live." The second speeds up the renaming process through an algorithmic improvement. The third and final improvement addresses a problem that arises in translating SSA form back into executable code. We have implemented all three improvements in our laboratory compiler. The paper presents measurements that demonstrate the impact of each improvement.

The remainder of this paper is laid out as follows. Section 2 provides an overview of SSA form and the algorithms for constructing it. Section 3 describes a simple and efficient improvement on the minimal form that drastically reduces the number of ϕ-functions without resorting to global data-flow analysis. Section 4 shows how to manage the data structures to improve the efficiency of the renaming phase of SSA construction. Section 5 describes a subtle problem that arises in converting SSA form back into executable code and presents an algorithm for handling it. Each section contains experimental results that assess the impact of each improvement.

2 Background

Many techniques for the analysis and optimization of compiled code rely on the construction of information chains, either from uses to definitions or from definitions to uses [15]. Modern compilers often use SSA form as a sparse alternative to classic information chains. Informally, the code for a procedure is said to be in SSA form if it meets two criteria:

1. each name has exactly one definition point, and

2. each use refers to exactly one name.

The first criterion creates a correspondence between names and definition points. The second criterion forces the insertion of new definitions at points in the code where multiple values, defined along different paths, come together.

To satisfy the first criterion, the compiler must rewrite the code by inventing new names for each definition and substituting these new names for subsequent uses of the original names. To create unique names, the compiler adds a subscript to the original program name. This retains information about the original name

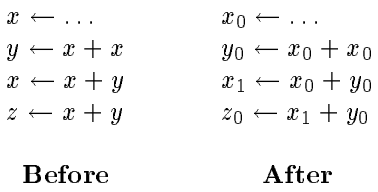

Figure 1 Straight-line code and its conversion to SSA form

space and improves the readability of the resulting code. To build SSA form from a straight-line fragment of code is trivial; each time a name gets defined, the compiler invents a new name which it then substitutes into subsequent references. At each re-definition of a name, the compiler uses the next new name and begins substituting that name. For example, consider the code in the left column of Figure 1. Conversion to SSA form produces the code in the right column. For straight line code, this process is quite simple and fast.

Renaming changes the availability of values. Consider the value of x_0 in the example. In the original code, x_0 was available at the definition of x_1, but not at the definition of z_0. In the SSA form, since x_0 and x_1 are distinct, both x_0 and x_1 can be named at the definition of z_0. The transformation correctly rewrote the last statement to refer to x_1 rather than x_0, but x_0 can still be referenced, in the sense that it has a unique name. Its value is accessible until the statement defining x_0 is re-executed.[1] In the original code, it could not be named.

The presence of control flow complicates both the renaming process and the interpretation of the resulting code. If a name in the original code is defined along two converging paths, the SSA form of the code has multiple names when it reaches a reference. To solve this problem, the construction introduces a new definition point at the merge point in the CFG. The definition uses a pseudo function, called a ϕ-function or a ϕ-node. The arguments of the ϕ-node are the names flowing into the convergence, and the ϕ-node defines a single, new name. Subsequent uses of the original name will be replaced with the new name defined by the ϕ-node. This ensures the second criterion stated earlier: each use refers to exactly one name. To understand the impact of ϕ-nodes, consider the code fragment shown in Figure 2. Two different definitions of x reach the use. The construction inserts a ϕ-node for x at the join point; it selects from its arguments based on the path that executes at run-time.

Conceptually, the SSA construction involves two steps. The first step decides where ϕ-nodes are needed. At each merge point in the CFG, it must consider, for each value, whether a ϕ-node is required and insert those for which the answer is affirmative. The second step systematically renames all the values to correspond to their definition points. For a specific definition, this involves rewriting the left-hand side of the defining statement and the right-hand side of every reference to the value. At a merge point, the value may occur as an argument to a ϕ-node. When this happens, the name propagates no further along that path. (Subsequent uses refer to the name defined by the ϕ-node.)

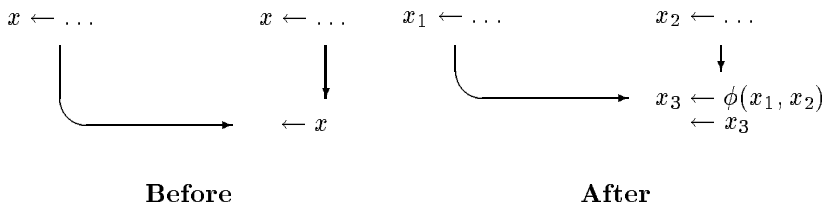

Figure 2 Conversion to SSA form in the presence of control flow

[1] This notion gives rise to the "static" in "static single assignment." The uniqueness of names is a static property rather than a dynamic property. A given assignment statement can execute more than once at run-time.

The simplest SSA conversion algorithm would insert a ϕ-node at each join point for each original name referenced in the procedure. Renaming would be done in two reverse-postorder passes; the first pass ignores back edges and the second pass rewrites only names that correspond to values passed along back edges. The resulting SSA form would be huge; it would have many more ϕ-nodes than necessary. However, it would conform to the two criterion. We might call this "maximal" SSA form. Although this algorithm is hopelessly inefficient, it captures the essence of the SSA construction process: decide where to place ϕ-nodes, then rewrite the name space. The difference between this algorithm and those that follow is optimization; all the other algorithms produce fewer ϕ-nodes and, consequently, smaller name spaces.

2.1 Building Minimal SSA

Figure 3 shows the basic algorithm for constructing minimal SSA form from a CFG representation of the routine. The algorithm has two basic steps: determine locations for ϕ-nodes and rename variables. The scheme for ϕ-node placement uses information about dominator relationships in the CFG to determine where ϕ-nodes are needed. The renaming step uses a preorder walk over the dominator tree and an array of stacks to introduce new names and track their appropriate scopes.

The first step in placing ϕ-nodes builds a *dominator tree* for the CFG and calculates *dominance frontiers* for the nodes in the CFG. In a flow graph, if node X appears on every path from the start node to node Y, then X *dominates* Y ($X \underline{\gg} Y$). If $X \underline{\gg} Y$ and $X \neq Y$, then X *strictly dominates* Y ($X \gg Y$). The *immediate dominator* of Y (idom(Y)) is the closest strict dominator of Y[13]. In the routine's *dominator tree*, the parent of each node is its immediate dominator. Notice that all nodes that dominate a node X are ancestors of X in the dominator tree. Lengauer and Tarjan give an efficient algorithm for building the dominator tree in $\mathbf{O}(E \log N)$ time, where E is the number or edges and N is the number of blocks in the CFG [17]. The *dominance frontier* of node X is the set of nodes Y such that X dominates a predecessor of Y, but X does not strictly dominate Y (*i.e.*, $\mathrm{DF}(X) = \{Y \mid \exists P \in \mathrm{Pred}(Y), X \underline{\gg} P \text{ and } X \not\gg Y\}$). Intuitively, DF($X$) is the set of nodes one edge beyond the region that X dominates, and thus identifies blocks reached on different control-flow paths. Cytron *et al.* give an algorithm for finding dominance frontiers which runs in $\mathbf{O}(E + N^2)$ time [10, Figure 10][2]. Cytron *et al.* extend the concept of diminance frontiers in two ways.

1. The dominance frontier of a set of nodes is defined to be the set of nodes in the dominance frontier of any member of the set (*i.e.*, $\mathrm{DF}(\mathcal{S}) = \cup_{X \in \mathcal{S}} \mathrm{DF}(X)$).
2. The *iterated* dominance frontier $\mathrm{DF}^+(\mathcal{S})$ is the limit of the sequence

$$\begin{aligned} \mathrm{DF}_1 &= \mathrm{DF}(\mathcal{S}) \\ \mathrm{DF}_{i+1} &= \mathrm{DF}(\mathcal{S} \cup \mathrm{DF}_i) \end{aligned}$$

For each variable v, the compiler builds a set $\mathcal{A}(v)$ containing the CFG nodes where assignments to v occur. Cytron *et al.* show that ϕ-nodes for v are required only in blocks in $\mathrm{DF}^+(\mathcal{A}(v))$. To improve the efficiency of ϕ-node placement, both Cytron and Ferrante and Sreedhar and Gao have proposed more efficient schemes [11, 19]. The improvements that we propose in the following sections are also effective in these frameworks.

After ϕ-nodes have been inserted, variables must be renamed to create the single-assignment property. This is accomplished in a single recursive walk of the dominator tree, shown in the procedure `SEARCH` in Figure 3. For each name in the original code, `SEARCH` maintains two data structures. The first, *Counters*[v], contains the subscript that will be assigned to the next definition of v. The second, *Stacks*[v], holds the current subscript for v. At each definition of v, `SEARCH` renames v with the subscript from *Counters*[v], pushes that value onto *Stacks*[v], and increments *Counters*[v]. During the first step, it rewrites variable names, incrementing the various counters and pushing new names onto the appropriate stacks. Next, it rewrites ϕ-node parameters in any successor blocks in the CFG so that the name inherited from the current block has the current subscript. (It uses the `whichPred` function to determine which ϕ-node parameter in the successor corresponds to the current block.) To continue the search, it recurses on each child in the dominator tree. On return from the recursion, it processes the current block again, to pop from each stack any subscripts added while processing the block.

[2]Cytron *et al.* assert that the N^2 only occurs in the worst case and that the figure is closer to N in practice

```
/* STEP 1: Determine locations for φ-nodes */
Calculate the dominator tree and dominance frontiers

For each variable, v
    A(v) ← {blocks containing an assignment to v}
    Place a φ-node for v in the iterated dominance frontier of A(v)


/* STEP 2: Rename each variable, replacing v, with the appropriate v_i */
For each variable, v
    Counters[v] ← 0
    Stacks[v] ← emptystack()
SEARCH(start)


/* Recursively walk the dominator tree, renaming variables */
SEARCH(block)
    For each φ-node, v ← φ(...), in block
        i ← Counters[v]
        Replace v by v_i
        push(i, Stacks[v])
        Counters[v] ← i + 1
    For each instruction, v ← x op y, in block
        Replace x with x_i, where i = top(Stacks[x])
        Replace y with y_i, where i = top(Stacks[y])
        i ← Counters[v]
        Replace v by v_i
        Push i onto Stacks[v]
        Counters[v] ← i + 1
    For each successor, s, of block
        j ← whichPred(s, block)
        For each φ-node, p, in s
            v ← j^th operand of p
            Replace v with v_i, where i = top(Stacks[v])
    For each child, c, of block in the dominator tree
        SEARCH(c)
    For each instruction, v ← x op y, or φ-node, v ← φ(...), in block
        pop(Stacks[v])
```

Figure 3 Algorithm for building minimal SSA form

2.2 Building Pruned SSA

Minimal SSA form relies entirely on dominator information to determine where to insert ϕ-nodes. The dominance frontier correctly captures the potential flow of values, but ignores the data-flow facts themselves – in particular, knowledge about the lifetimes of values gleaned from analyzing their definitions and uses. Because of this, the minimal SSA construction will insert a ϕ-node for v at a join point where v is not live.

To improve on minimal SSA, Cytron *et al.* proposed another variation on SSA that they called *pruned* SSA [10]. To build pruned SSA, the compiler first performs "liveness analysis" on the routine. Liveness analysis produces, for each block, a set of values that are live on entry to the block – that is, values that can be referenced along some path leading to the block [1]. Many algorithms exist for computing live information [16].

The actual construction of pruned SSA is quite similar to the construction of minimal SSA. In Figure 3, we need only add a prepass that computes live information and modify the first step where ϕ-nodes are inserted. The minimal SSA construction inserts a ϕ-node for v in *every* node in $\mathrm{DF}^+(\mathcal{A}(v))$. The pruned SSA construction changes this to insert a ϕ-node for v in every node $n \in \mathrm{DF}^+(\mathcal{A}(v))$, where $v \in \textsc{live\_in}(n)$. These changes can drastically reduce the number of ϕ-nodes.

The pruned-SSA construction algorithm costs more than the minimal SSA construction. Not only does inserting ϕ-nodes require two membership tests rather than one, but it must also compute the LIVE sets. Although linear-time or near-linear time algorithms exist for this problem [14, 12, 22] (and, thus, the asymptotic time complexity of SSA conscruction does not change), it does raise the constant factor substantially. To compute LIVE sets, the analyzer must make a pass over each block to build sets containing the initial information. Then, in a second step, it revisits each block to compute the actual LIVE sets.[3] These operations consume a nontrivial amount of time.

Equally troubling, building liveness analysis increases the space requirements for building SSA, since each block has a number of large sets associated with it. These larger memory requirements can directly degrade performance.

2.3 One final assumption

Throughout this paper, we assume that names are used in a type-consistent fashion. A name in the original code cannot be used to hold values that have different types, such as an `integer` along one path and a `float` along another. This is true in most modern programming languages. It becomes somewhat trickier when the input program is at a very-low level. For example, building SSA on code produced by a register allocator is problematic if a single register can hold either an integer or a floating point value. The construction algorithms implicitly assume that they can determine the type of a ϕ-function from its inputs. If its inputs have different types, the assumption is violated.

```
non-locals ← ∅
For each block B
    killed ← ∅
    For each instruction v ← x op y in B
        if x ∉ killed then
            non-locals ← non-locals ∪ {x}
        if y ∉ killed then
            non-locals ← non-locals ∪ {y}
        killed ← killed ∪ {v}
```

Figure 4 Algorithm for finding non-local names

[3]The number of "visits" to each block will depend on the specific data-flow analysis algorithm used and on the detailed structure of the routine being analyzed.

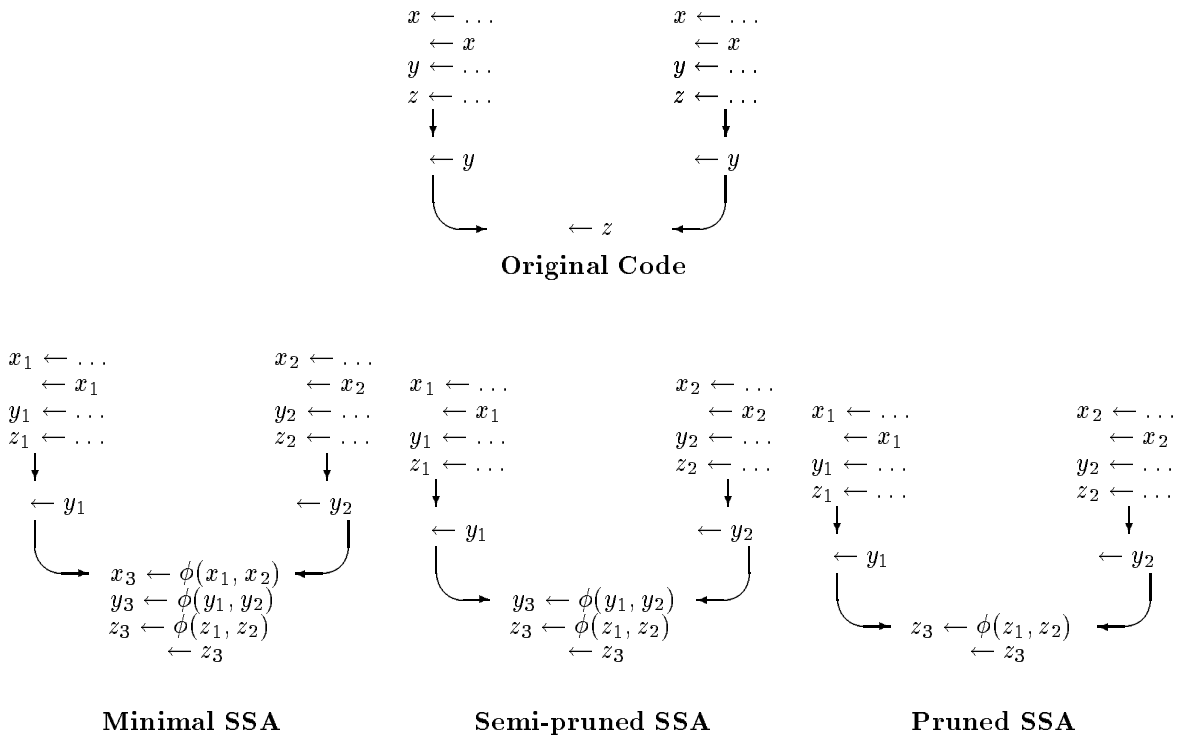

Figure 5 Three flavors of SSA form

3 Using Fewer ϕ-Nodes — Semi-Pruned Form

Cytron *et al.* describe two flavors of SSA form that vary in the number of ϕ-nodes inserted. *Minimal* SSA form places ϕ-nodes by looking only at the dominance frontier information without regard to liveness. In other words, it is possible that a ϕ-node will be inserted for a name which is not subsequently used. These extra ϕ-nodes do not detract from the quality of analysis; they simply waste space and time. *Pruned* SSA form relies on *liveness analysis* to ensure that no such dead ϕ-nodes are inserted. If we are building pruned SSA form, we only insert a ϕ-node for a variable v at the beginning of a block if v is live on entry to that block. Since the pruned form relies on additional analysis, it may be slower to build. However, the time spent on analysis may be recovered by inserting fewer ϕ-nodes.

We have developed a third flavor of SSA that we call *semi-pruned* SSA form. The speed and space advantage of this form over the other two relies on the observation that many names in a routine are defined and used wholly within a single basic block. For example, the compiler typically generates temporary names to hold intermediate steps in any non-trivial computation; these compiler-generated names often have short lifetimes. Semi-pruned SSA capitalizes on this observation by computing the set of names that are live on entry to *some* basic block in the program. We call these "non-local" names. The construction only computes $\mathcal{A}(v)$ for non-local names. The number of ϕ-nodes will lie between that of the minimal and pruned forms, but the non-local names are much cheaper to compute than the full-blown liveness analysis. Therefore, semi-pruned form represents a compromise between the time required to perform liveness analysis and the reduction in the number of ϕ-nodes that it allows.

To discover non-local names, the construction uses the algorithm shown in Figure 4. The compiler makes a simple forward pass over each basic block. When it discovers an operand that has not already been defined within the block (the *killed* set), it must be a non-local name. Notice how much simpler this is than performing the complete live analysis required for the pruned SSA construction. Computing non-local names requires just two sets, *non-local* and *killed* – much less space than the three sets per block required for a full live analysis. The algorithm makes just one pass over each block; this avoids the overhead for either

iteration or elimination in a full data-flow analysis. The *non-local* set is initialized once; the *killed* set is reset for each block[4]. The time and space requirements for building *non-local* are, therefore, minimal.

Figure 5 illustrates the differences between the three flavors of SSA. In the original code, we define three variables, x, y, and z. The three graphs at the the bottom of the figure compare the ϕ-nodes which the three flavors of SSA insert. The minimal SSA form contains ϕ-nodes for all three variables. Clearly, the ϕ-nodes for x and y are unnecessary; these variables are never used again. The semi-pruned SSA form does not contain a ϕ-node for x because it is not live across any basic-block boundary. However, we still insert a ϕ-node for y, because it is live across *some* block boundary, and that is the limit of the analysis used. The pruned SSA form contains a ϕ-node for z only. For pruned form, we performed the complete analysis necessary to show that both x and y are never used again.

Each of the above three flavors has different uses. Cytron *et al.* give a contrived example where global value numbering might benefit from the extra, dead ϕ-nodes which minimal form contains. However, these dead ϕ-nodes constitute a waste of both time and space for optimizations like constant propagation and dead-code elimination, which use the semi-pruned form. Other optimizations, such as the *peephole optimizer* and the *register allocator*, depend on the compactness of the pruned form and so must bear the extra time needed to perform the required liveness analysis [8, 5].

3.1 Experimental Results

We compared the various flavors of SSA using routines from the SPEC benchmark suite [18]. Table 1 shows the number of ϕ-nodes and the time required to build the three flavors for each routine. The number of ϕ-nodes required by semi-pruned form always falls between that of minimal and pruned. However, the time required to build semi-pruned form is often shorter than the time required for either minimal or pruned. This is due to the effective compromise between the fast analysis and a reduction in the number of ϕ-nodes inserted. We also compare the time required by global value numbering (after the code is in SSA form) for each of the three flavors [2]. This algorithm requires $\mathbf{O}(E \log N)$ time, where N and E are the number of nodes and edges in the SSA graph.[5] The experiments show that reducing the number of ϕ-nodes can *?significantly?* improve the execution time of this analysis.

4 Efficient Stack Manipulation

In the second step of the SSA construction (see Figure 3), the compiler renumbers all the names to ensure that each assignment (including ϕ-nodes) defines a unique name. The renumbering is handled by a recursive preorder walk over the dominator tree.

We can summarize the renaming process as follows: We declare an array of stacks (indexed by the original name) to hold the subscripts used to replace each original name, and we use the topmost name on the stack

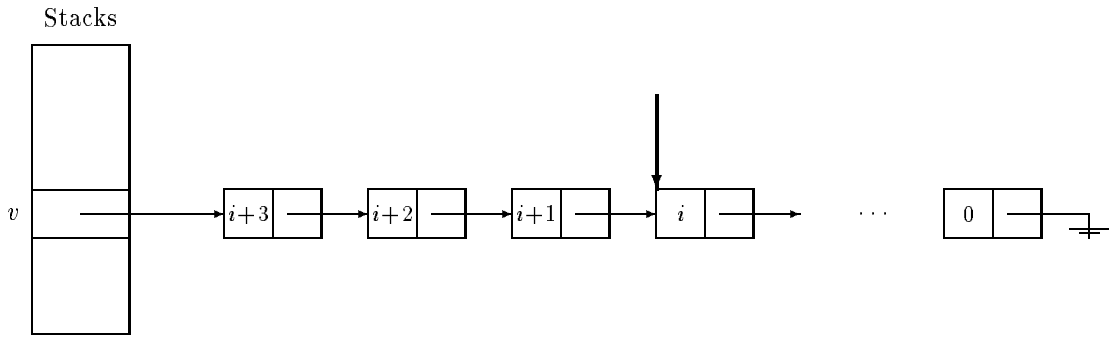

Figure 6 Stacks after variable v is defined three times

[4]Further, by utilizing the *SparseSet* data structure[7], the time to perform these actions is constant.

[5]In the SSA graph, each node represents an assignment and edges flow from uses to definitions.

	Number of ϕ-nodes			*Time to build SSA (sec)*			*Value numbering (sec)*		
Routine	*Minimal*	*Semi*	*Pruned*	*Minimal*	*Semi*	*Pruned*	*Minimal*	*Semi*	*Pruned*
`twldrv`	73778	11989	9886	1.28	0.34	0.45	8.42	4.88	4.75
`deseco`	8610	2216	1842	0.23	0.18	0.22	1.92	1.58	1.46
`ddeflu`	5852	1560	1222	0.12	0.06	0.09	0.77	0.63	0.62
`iniset`	5364	1080	462	0.12	0.11	0.16	0.34	0.19	0.17
`debflu`	4715	1748	1542	0.09	0.08	0.08	0.85	0.72	0.71
`paroi`	3597	767	632	0.07	0.06	0.07	0.35	0.22	0.22
`efill`	3170	357	74	0.04	0.02	0.02	0.14	0.04	0.04
`inisla`	2722	267	141	0.04	0.03	0.03	0.12	0.05	0.05
`tomcatv`	2699	365	145	0.05	0.03	0.05	0.14	0.06	0.05
`pastem`	2584	374	62	0.06	0.03	0.05	0.20	0.10	0.10
`prophy`	2021	436	401	0.05	0.04	0.05	0.36	0.27	0.31
`inithx`	1967	267	85	0.04	0.03	0.04	0.22	0.16	0.14
`debico`	1880	171	112	0.04	0.03	0.04	0.14	0.08	0.08
`repvid`	1094	141	45	0.03	0.02	0.03	0.07	0.04	0.03
`bilan`	1080	70	34	0.03	0.02	0.02	0.10	0.06	0.06
`dyeh`	857	79	40	0.02	0.02	0.02	0.05	0.03	0.02
`sgemm`	809	341	279	0.02	0.01	0.01	0.05	0.05	0.04
`orgpar`	803	143	98	0.03	0.02	0.03	0.09	0.05	0.06
`integr`	799	89	34	0.02	0.01	0.02	0.05	0.02	0.01
`gamgen`	761	85	39	0.02	0.01	0.01	0.04	0.02	0.03
`heat`	667	50	22	0.02	0.02	0.02	0.05	0.03	0.02
`fmtgen`	653	127	33	0.01	0.01	0.01	0.02	0.02	0.01
`inideb`	645	148	131	0.01	0.01	0.01	0.07	0.05	0.05
`yeh`	624	154	122	0.02	0.02	0.03	0.09	0.09	0.09
`drepvi`	617	76	52	0.03	0.02	0.02	0.03	0.04	0.03
`cardeb`	601	96	54	0.02	0.01	0.01	0.03	0.02	0.02
`ihbtr`	597	88	31	0.01	0.01	0.02	0.04	0.02	0.01
`bilsla`	569	67	38	0.01	0.01	0.01	0.03	0.02	0.01
`drigl`	557	169	121	0.01	0.01	0.01	0.04	0.04	0.04
`saturr`	541	27	25	0.03	0.02	0.03	0.08	0.06	0.06
`dcoera`	334	36	33	0.01	0.01	0.01	0.03	0.01	0.02
`lissag`	311	42	15	0.01	0.01	0.01	0.02	0.01	0.01
`colbur`	310	15	9	0.01	0.01	0.01	0.02	0.02	0.01
`fmtset`	275	77	61	0.01	0.01	0.01	0.03	0.02	0.01
`supp`	38	7	5	0.02	0.03	0.03	0.07	0.05	0.06
`fpppp`	0	0	0	0.17	0.21	0.30	0.80	0.80	0.77
`subb`	0	0	0	0.02	0.02	0.02	0.05	0.06	0.05

Table 1 Comparison of three flavors of SSA

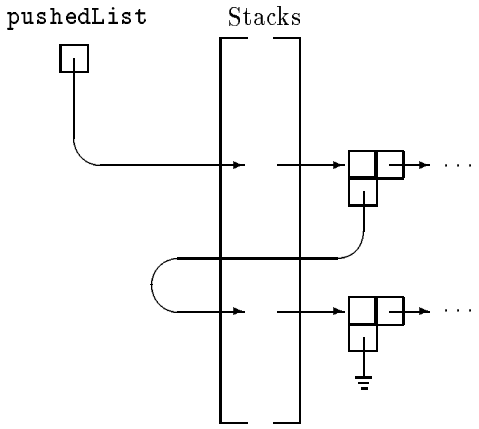

Figure 7 The stacks data structure, showing the connection between nodes in each stack

to annotate each use of that name. We push a new subscript onto a name's stack each time we encounter a definition of that name.

When we have finished processing a block (and its descendants in the dominator tree), we must restore the stacks to the same state as when we began processing the block. The method suggested by Cytron *et al.* is to iterate a second time through the current block's ϕ-nodes and instructions, this time popping a name from the appropriate stack for each definition. However, pushing a node for each definition in a block is wasteful. Consider a basic block that defines a variable v three times. Figure 6 shows the stack for v after processing the definitions in the block. The vertical arrow indicates the point where we must restore the stack. Notice that once the $i+2$ node gets pushed, the $i+1$ node can never be accessed again, because any subsequent reference to v will use the name in the $i+2$ node and restoring the state of the stack will remove the $i+1$ node. Similarly, after the $i+3$ node gets pushed, the $i+2$ node will never be accessed again. We can reduce the number of nodes allocated if we *overwrite* the $i+1$ node with $i+2$ and then with $i+3$. However, we cannot overwrite the i node, because it must remain after we have restored the state of the stack. Therefore, we will push a node onto the stack for v at the *first* definition of v in the block, but subsequent definitions of v in the same block will simply overwrite the node. To accomplish this, we will record which variables have already had a node pushed for the current block. If a variable is redefined inside the block, we will overwrite its top-of-stack instead of pushing a new node.

Since we are pushing at most one node for each variable when we process the definitions in a block, we can no longer restore the state of the stack by iterating over the operations in the block and popping a node for each definition. For each block, we maintain a list of the variables with a node that has been pushed. Nodes are added to the list as they are pushed (*i.e.,* after the first definition in the block). Thus, restoring the state of the stacks requires popping the nodes in the list. This data structure is shown in Figure 7.

In summary, we must ensure that at most one node per variable gets pushed per block, and we use a list to guide the popping of the stacks. This improvement not only keeps us from allocating superfluous nodes, but it also speeds up the popping phase at the end. The approach used by Cytron *et al.* requires a second pass through the instructions in the block, popping a node from each definition's stack as it is encountered. With this new method, we can simply iterate down the list of elements, popping just one node from each stack.

4.1 Experimental Results

Figure 2 compares the number of pushes required for each method of manipulating the stacks. The *old method* performs a push for each definition in the routine, but the *new method* performs at most one push per variable per block. The number of pushes (and, thus, the amount of memory required) is significantly reduced when the new method is used. We also compare the total time required to build semi-pruned SSA form using each of the methods. For large routines, a *?significant?* amount of time is saved.

	Number of pushes		*Build SSA (sec)*	
Routine	*Old method*	*New method*	*Old method*	*New method*
twldrv	27295	19569	0.350	0.320
fpppp	19963	5641	0.270	0.240
deseco	14121	8625	0.190	0.160
iniset	6608	5298	0.110	0.100
ddeflu	6393	4651	0.080	0.070
debflu	6389	4225	0.080	0.060
paroi	4881	2864	0.050	0.070
prophy	3609	1947	0.050	0.030
pastem	2755	2060	0.050	0.030
inithx	2686	1706	0.040	0.030
debico	2667	1348	0.030	0.030
tomcatv	2633	1490	0.040	0.030
inisla	2373	1308	0.030	0.030
supp	2037	1734	0.020	0.040
bilan	1994	878	0.030	0.020
subb	1733	1311	0.020	0.020
saturr	1689	1522	0.020	0.020
drepvi	1597	1109	0.030	0.020
yeh	1547	1157	0.030	0.010
orgpar	1499	1053	0.020	0.020
repvid	1449	1010	0.020	0.020
efill	1439	1047	0.010	0.020
inideb	1242	729	0.010	0.020
heat	944	845	0.010	0.010
sgemm	941	681	0.010	0.020
dyeh	941	794	0.010	0.020
cardeb	893	643	0.010	0.010
gamgen	849	417	0.010	0.010
drigl	805	648	0.010	0.010
integr	804	499	0.010	0.010
bilsla	750	339	0.010	0.010
ihbtr	732	605	0.010	0.010
lissag	724	292	0.010	0.000
colbur	716	538	0.010	0.010
fmtset	668	447	0.010	0.010
fmtgen	635	548	0.010	0.010
sortie	595	503	0.010	0.010
dcoera	556	455	0.010	0.000
sgemv	348	269	0.010	0.000
coeray	348	319	0.010	0.010
sigma	129	126	0.010	0.010
arret	63	62	0.010	0.000
vgjyeh	60	54	0.010	0.000

Table 2 Comparison of stack handling methods

5 Replacing ϕ-nodes with Copies

After optimization, the compiler must translate the SSA form of a routine back into an executable form. We know of no popular computer that has a hardware ϕ-function; thus, the compiler must translate the semantics of the ϕ-function into commonly implemented instructions. Ideally, this translation would restore the name space used in the original code. Often, however, optimization and translation have made this impossible; in such cases, the compiler must insert copy operations to mimic the actions of the ϕ-function.

To replace a ϕ-node in block b, the compiler can insert a copy operation into each of b's predecessors. Since the meaning of a ϕ-node is a mapping of all of the incoming values to a single name, n, it is equivalent to place a copy at the end of each predecessor block. The copy moves the value corresponding to the appropriate ϕ-node parameter into n. Consider the example in Figure 8. The left-hand side of the figure shows a fragment of the CFG with the code in SSA form. The right-hand side of the figure shows the same fragment with copies inserted for the ϕ-node. Note that the insertion of copy operations has made the ϕ-node obsolete, so we can discard it. This process can produce a large number of copies; in our compiler, we rely on the coalescing phase of a graph-coloring register allocator to remove as many of these as possible [8, 5].

The process of inserting copies for ϕ-nodes will consist of iterating through the blocks in the CFG, inserting a copy for each parameter of each ϕ-node in the predecessor of the block containing the ϕ-node. This process, however, is anything but straightforward – the interaction of names along different paths in the CFG can lead to subtle errors. We will first present two examples which require a more sophisticated approach than the naive insertion of copies, and then we will present our algorithm for replacing ϕ-nodes.

5.1 Refining the copy-insertion algorithm

We can insert a copy operation at the end of a block for each of its successors' ϕ-nodes. This is a straightforward instruction insertion for each of the successors' ϕ-nodes. However, this naive insertion of copies can cause errors in cases that involve copy-folding in the renaming phase[6] and critical edges in the control-flow graph.

Copy Folding Folding copies reduces the size of the name space and simplifies the SSA graph. During the renaming phase of the SSA construction, the compiler can perform copy folding in a particularly simple and elegant manner. This can speed both analysis and optimization. To perform copy folding, the compiler interprets a copy as an operation on the name stacks; at a copy $v_i \leftarrow x_j$, it pushes the name x_j onto the name stack for v. This ensures that the compiler rewrites subsequent uses of v_i to refer directly to x_j.

Critical Edges A *critical edge* is defined as an edge between a block with multiple successors and a block with multiple predecessors (*i.e.*, (i, j) is a critical edge if and only if $|\text{succ}(i)| > 1$ and $|\text{pred}(j)| > 1$). On a critical edge, the copy insertion described above breaks down. The copy cannot be inserted into the edge's source (the predecessor), because it would execute along paths not leading to the ϕ-node. Similarly, it cannot be inserted in the edge's sink (the successor), because it would destroy values coming from other predecessors.

This problem can be addressed by *splitting* the critical edge – inserting an empty basic block along the edge. Figure 9 shows a critical edge and how it could be split. In the presence of certain control-flow operations (*e.g.*, JUMP-REGISTER), it is not always possible to split critical edges. Similarly, in the late stages of compilation, particularly instruction scheduling, splitting the edge may be impractical. Critical edges are important for code placement algorithms, because their presence can restrict the movement of code, and they can also cause naming conflicts when replacing ϕ-nodes with copies.

5.1.1 The "Lost-Copy" Problem

The *lost-copy problem* can only occur when copies are folded, *and* when critical edges have not been split. This situation requires care not only in the method of inserting the copies into a block, but also the order in which we iterate through the blocks.

[6] The same naming problems caused by copy folding can also occur if an optimization pass performs aggressive renumbering, as is done by a value numberer, for example.

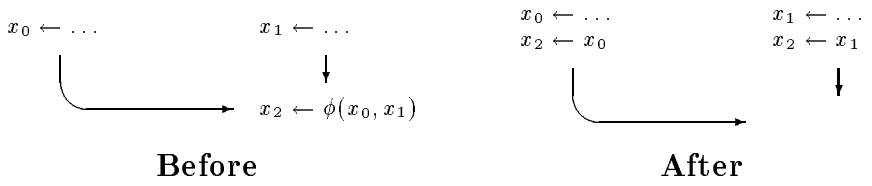

Figure 8 The impact of inserting copies for ϕ-nodes

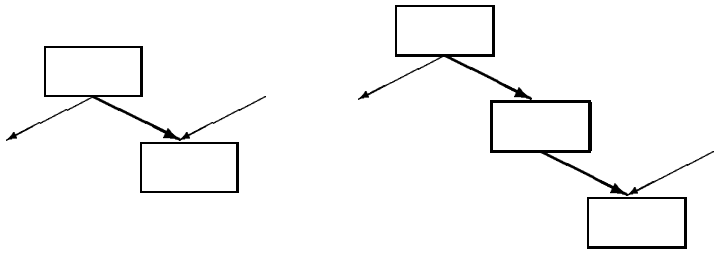

Figure 9 Splitting a critical edge

Consider the code on the left side of Figure 10. At each iteration, the loop increments a variable, and the value from the penultimate iteration is then returned[7]. The second column shows the code translated into SSA form with copy folding. Notice how y disappeared. The third column shows the result of naively replacing the ϕ-node with copies. Clearly, the result of the code has changed; it now prints out the value of the last iteration. The final column shows how splitting the critical edge cures the problem.

Intuitively, the naive copy insertion failed because it created a reference to x_2 beyond the scope of the ϕ-node that defined it. The `RTN` occurs after the definition of x_3; x_2 and x_3 are related values. Folding x_2 for y extends the lifetime of x_2 beyond the redefinition that creates x_3.

To avoid this problem, the compiler must notice that the value overwritten by the new copy is live past the point where the copy is inserted. When it detects this situation, it can insert a copy to a new temporary name prior to inserting the copy, and rewrite subsequent uses of the overwritten name with the temporary's

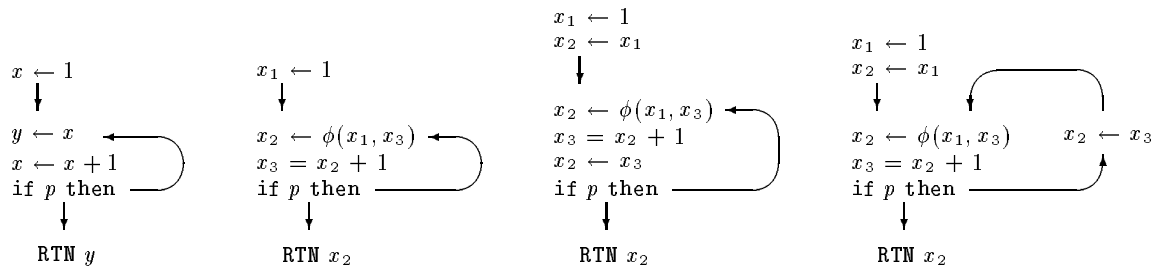

Figure 10 An example of the code leading to the "Lost-Copy" problem

[7]While this example might seem contrived, the situation arises routinely in Fortran DO-loops.

```
Perform liveness analysis
For each variable v
    Stacks[v] ← emptystack()
insert_copies(start)

insert_copies(block)
    pushed ← ∅
    For all instructions i in block
        Replace all uses u with Stacks[u]

    schedule_copies(block) /* see Figure 14 */
    For each child c of block in the dominator tree
        insert_copies(c)
    For each name n ∈ pushed
        pop(Stacks[n])
```

Figure 11 Algorithm for iterating through the blocks to perform ϕ-node replacement

---

name. This is the idea underlying our copy insertion algorithm. This rewriting with new names mimics the name rewriting phase in the SSA construction, implying that the compiler must walk the dominator tree to insert copies. It also means that the implementation will require a stack of names similar to the *Stacks* used when building SSA form. However, copy insertion only need to push names onto stacks corresponding to the names defined by the inserted copies – these are the only names which need to have their uses rewritten.

The algorithm uses live-out information to determine which registers require insertion of additional copies to temporaries. It uses a structure like the *Stacks* array to record the newly-created temporary names. This results in an algorithm that walks the dominator tree in preorder. For each block, it replace uses in ϕ-nodes and instructions with any new names. Next, it builds a list of copies that must be inserted and uses the algorithm outlined in Section 5.1.2 to determine the order to insert the copies. As each copy is inserted, if its source is live at the end of the block, the algorithm pushes the destination name onto the source's stack and resets a flag to show that the source is live outside the block. Finally, if the destination of the copy to be inserted is live past the end of the block, it inserts a copy to a temporary at the ϕ-node which defines the register.

Some clever engineering is also required to make this as efficient as possible. A block B can be the predecessor to many other blocks, but imagine the case where each of the successor blocks requires a copy to its own temporary for some value flowing out of B. A naive implementation would insert as many copies to temporaries as B has successors. One solution to this problem is to insert a copy to a temporary (when it is needed) at the top of the block to which the current ϕ-node is attached and to use this temporary's name whenever the value is needed as the source of a copy. This has the practical effect of capturing the value in question immediately after it is defined by the ϕ-node, so that it cannot be overwritten. Other solutions exist, but their effect on code size is unpredictable.

The algorithm for inserting copies for ϕ-nodes which avoids the lost-copy problem is shown in Figure 11. Notice that the code must be in the form of a recursive routine to perform the walk. Clearly, the algorithmic complexity is bounded by the liveness analysis rather than this walk over the CFG.

5.1.2 The "Swap" Problem

Copy folding exposes another problem with the naive copy insertion algorithm. Figure 12 shows an example. We refer to this as the *swap problem*.

The left side of the figure shows a simple loop that swaps the values of two variables using a temporary named x. The middle column shows the SSA form after folding copies. Since all of the operations in the body of the loop were copies, they have all been absorbed, and all that is left in the body are the ϕ-nodes.

The right side of Figure 12 shows the result of naively inserting copy operations for the ϕ-nodes. This code is clearly incorrect. On the first iteration of the loop, the value of a_2 gets overwritten, and both a_2 and b_2 subsequently contain the same value. The problem stems from the fact that the ϕ-nodes in a block are

considered to execute in parallel. To solve this problem, the compiler can introduce a temporary variable for each copied value.

Naively inserting copies of all values into temporaries, however, is not a feasible solution. It potentially doubles the number of copies necessary for ϕ-node replacement. Instead, the compiler should insert the minimal number of copies to temporaries necessary for correctness. Consider again the example in Figure 12. The problem is that some of the parameters to the ϕ-nodes are defined by other ϕ-nodes in the same block. Notice that the copies inserted into the top block do not contain references to other names defined by a ϕ-node. These copies have been inserted correctly – that is, they do not change the meaning of the code. It is only the copy operations inserted for *parameters which were themselves defined by ϕ-nodes in this block* which caused the problem. Thus, inserting copies to temporaries for these special cases will produce correct code.

This is slightly simplistic, however. Consider the code in Figure 13. Here, there is not a cycle of dependences as in the swap problem, although the name a_2 *is* used in a successive ϕ-node in the block. According to the above rule, since the ϕ-node is used as a parameter in another ϕ-node in that block, a copy to a temporary should be inserted for it. Simple analysis, though, will show that reordering the copies will produce correct code without the addition of a temporary, as shown in the right side of this figure.

In some sense, the choice of how to insert copy operations for ϕ-nodes and when to insert copies to temporaries is a scheduling problem. A copy operation has two arguments, the source and the destination. We want to insert copies for a set of ϕ-nodes subject to the following restriction: to schedule a copy c, all other copy operations which include c's destination as their source must be scheduled first. That is, before a name is overwriten, any other operation which needed its value must have it already.

Another way to look at this problem is to model the interaction of the copies to be inserted as a graph whose nodes represent the copies and whose edges represent a name defined by one copy and used in another copy. If the graph is acyclic, the schedule of copies can then be found by a simple topological sort of the graph – although we do not actually need to build this graph if we are careful about the data structures we use to build the schedule.

Our algorithm makes three passes over the list of ϕ-nodes. In the first pass, the compiler counts the number of times a name is used by other ϕ-nodes. In the second pass, it builds a worklist of names that are not used in other ϕ-nodes. The third pass iterates over the worklist, scheduling a copy for each element in the worklist. Obviously, the copy operations whose destinations are not used by other copy operations can be scheduled immediately. Furthermore, each time the compiler inserts a copy operation, it can add the source of that operation to the worklist.

Consider a block where the name n is used as the source for five other copy operations. By the rule given above, a copy redefining n cannot be inserted until all of the other five copies that use n have been inserted. The rule's intent is to ensure that all of the copy operations refer to the value of n before it is overwritten. But, once the first copy has been inserted, n's value has been preserved in its destination d, and overwriting n will not destroy that value. If the four remaining copy operations refer to d rather than n, then the compiler is free to overwrite n.

This tactic will ensure that the copy operations are ordered correctly, but it still does not address the problem of cycles of dependence. In the swap problem, we have a set of copies where each of the destinations

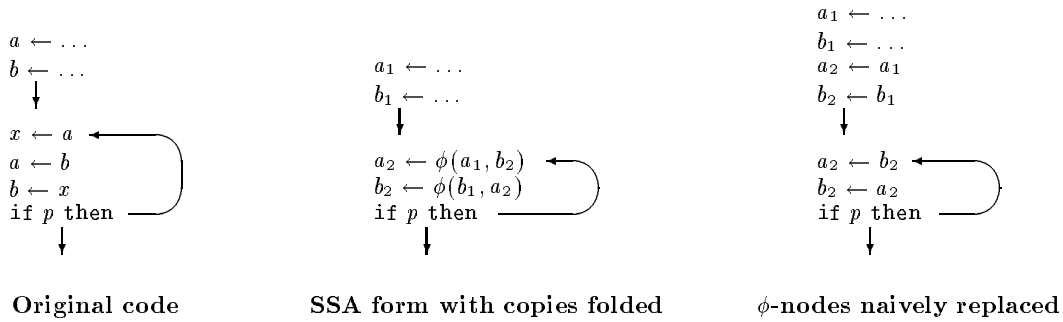

Figure 12 An example of the code leading to the "Swap" Problem

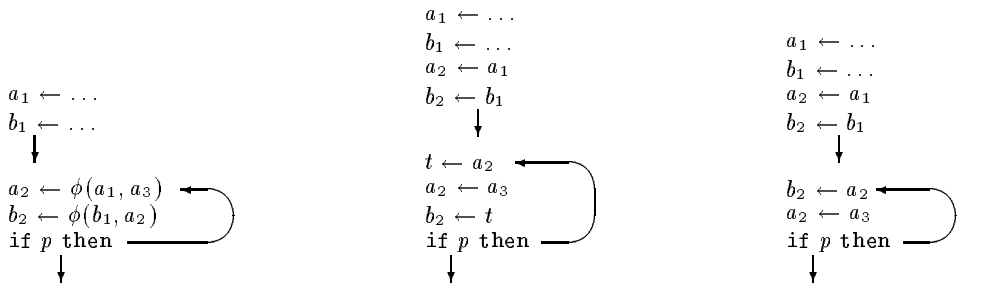

Figure 13 Simple ordering example

is used as a source in another copy in the set, forming a cycle. In the algorithm described thus far, none of the copies would ever be put on the worklist. To break this cycle, the algorithm can randomly pick one of the edges and break it, by inserting a copy to a temporary for one of the destinations. As we pointed out in the previous paragraph, this allows the algorithm to put that copy onto the worklist, and (with the cycle broken) schedule the rest of the copies.

The algorithm for solving the swap problem is shown in Figure 14. It is applied to each block and has three steps. The first step builds a list of the copies to be inserted by running through the ϕ-nodes in each of the block's successors. During this accumulation phase, it also records some facts, such as which destinations of the copies to be inserted are used as the sources of other copies in the list. The second pass builds up a worklist of those copies whose destinations are not used in any copies. The third step iterates through the worklist, inserting a copy for each member and then removing that member.

Each time a copy is removed from the worklist, its source is checked to see if it is a destination of another copy in the set of copies yet to be inserted. If so, it adds that new copy to the worklist. Even if this new copy is used as the source for numerous other copies waiting to be inserted; this is safe – remember that this algorithm is concerned with preserving values. Each time it inserts a copy, it records that the value formerly held in the source is now held in the destination. Any subsequent reference to the source in any inserted copy will use the destination's name instead of the source's name. Thus, it is free to overwrite the source after it copies the value into another location.

Whenever the compiler inserts a copy, it must also consider the lost-copy problem. Thus, before it inserts a copy, it must check to see if the destination is in the the block's live-out set. If it is, the compiler first inserts a copy of the destination's value to a temporary. Then, it pushes the temporary's name onto the *Stacks*. Subsequent blocks dominated by the current block will use the temporary's name in place of references to the destination's name.

We can summarize the process as follows. During the first step of this algorithm, the compiler built up the list of copies that needed to be inserted. Any copies left on this list when the worklist clears are involved in cycles. We know that at least one temporary will then need to be inserted, so the algorithm randomly picks one of the destination names to copy into a temporary name. This allows that copy to be put onto the worklist – the value is safely stored, so the compiler can overwrite the name. This breaks the cycle, and the the worklist-clearing loop can start again. It alternates between these two sections until all of the copies in the original list have been inserted.

6 Conclusions

The discovery of SSA form has revolutionized thinking and implementation of optimization. This paper has examined the implementation details in greater detail than the seminal literature on this subject.

The first half of this paper should serve as a survey of the different forms of SSA, wherein we presented a discussion of how to build each flavor, including the new semi-pruned form. This form is a compromise between the time required for liveness analysis required by pruned form and the large number of dead ϕ-nodes

```
schedule_copies(block)
    /* Pass One: Initialize the data structures */
    copy_set ← ∅
    For all successors s of block
        j ← whichPred(s, block)
        For each φ-node dest ← φ(...) in s
            src ← j^th operand of φ-node
            copy_set ← copy_set ∪ {⟨src, dest⟩}
            map[src] ← src
            map[dest] ← dest
            used_by_another[src] ← TRUE

    /* Pass Two: Set up the worklist of initial copies */
    For each copy ⟨src, dest⟩ in copy_set
        If used_by_another[dest] ≠ TRUE
            worklist ← worklist ∪ {⟨src, dest⟩}
            copy_set ← copy_set − {⟨src, dest⟩}

    /* Pass Three: Iterate over the worklist, inserting copies */
    While worklist ≠ ∅ or copy_set ≠ ∅
        While worklist ≠ ∅
            Pick a ⟨src, dest⟩ from worklist
            worklist ← worklist − {⟨src, dest⟩{
            If dest ∈ live_out_b
                Insert a copy from dest to a new temp t at φ-node defining dest
                push(t, Stacks[dest])
            Insert a copy operation from map[src] to dest at the end of b
            map[src] ← dest
            If src is the name of a destination in copy_set
                Add that copy to worklist
        If copy_set ≠ ∅
            Pick a ⟨src, dest⟩ from copy_set
            copy_set ← copy_set − {⟨src, dest⟩}
            Insert a copy from dest to a new temp t at the end of b
            map[dest] ← t
            worklist ← worklist ∪ {⟨src, dest⟩}
```

Figure 14 Algorithm for scheduling the copies to be inserted

found in minimal form. We also presented a more efficient method for manipulating the stacks used during the renaming phase of the SSA construction algorithm. Our algorithm reduces the number of nodes pushed and simplifies the process of popping nodes. The benefits and costs of each technique were discussed to give future implementors insights before they begin work.

The second half of the paper tackled the thorny problem of inserting copies for ϕ-nodes. We firmly believe that this a case of Backus' separation of concerns[3]. That is, each optimization pass should not be concerned with its impact on the final transformability of the ϕ-nodes, but, rather, the SSA transformer itself should have the ability to handle the code, regardless of the motion of instructions from a given optimization pass. This high ideal suffers from practical considerations, but we present an algorithmic solution to handle the problems.

When replacing ϕ-nodes with copies, we have found both the swap problem and the lost-copy problem in real world codes. Implementation of the special algorithms for inserting copies is essential to avoiding the incorrect code these two problems cause. We presented an algorithm that can efficiently insert ϕ-nodes in control-flow graphs without critical edges. When critical edges are present, we must perform a more complicated algorithm that includes liveness analysis and a preorder walk over the dominator tree in addition to the existing copy insertion algorithm.

7 Acknowledgements

This work was supported by DARPA through Army contract DABT63-95-C-0115 and by IBM through a graduate fellowship to L. Taylor Simpson. The work described in this paper has been done as part of the Massively Scalar Compiler Project at Rice University. The many people who have contributed to that project deserve our gratitude. Cliff Click of Motorola initially pointed out the swap problem to us. Bob Morgan of DEC served as a soundboard for some of our ideas on ϕ-node-directed copy insertion.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[3] John Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[6] Preston Briggs, KeithjD. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 00(00), June 1997. (*to appear.*

[7] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March–December 1993.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth 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.

[11] Ron K. Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995.

[12] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 22–34, Palo Alto, California, January 1975.

[13] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Programming Languages Series. Elsevier North-Holland, Inc., 52 Vanderbilt Avenue, New York, NY 10017, 1977.

[14] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[15] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3:163–179, 1978.

[16] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[17] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[18] SPEC release 1.2, September 1990. Standards Performance Evaluation Corporation.

[19] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, January 1995.

[20] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[21] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[22] F. Kenneth Zadeck. Incremental data flow analysis in a structured program editor. *SIGPLAN Notices*, 19(6):132–143, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*