SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(8), 859–881 (10 JULY 1998)

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

preston briggs[1], keith d. cooper[2], timothy j. harvey[2] and l. taylor simpson[3]
[1]*Tera Computer Company, 2815 Eastlake Ave. East, Seattle, WA 98102, USA*
(email: preston@tera.com)
[2]*Department of Computer Science, Rice University, 6100 S. Main - MS 132, Houston, TX 77005, USA*
(email: keith,harv@rice.edu)
[3]*Trilogy Development Group, 6034 W. Courtyard Dr., Austin, TX 78730, USA*
(email: Taylor_Simpson@triology.com)

SUMMARY

Static Single Assignment (SSA) form is a program representation that is becoming increasingly popular for compiler-based code optimization. In this paper, we address three problems that have arisen in our use of SSA form. Two are variations to the SSA construction algorithms presented by Cytron *et al.*[1] The first variation is a version of SSA form that we call 'semi-pruned' SSA. It offers an attractive trade-off between the cost of global data-flow analysis required to build 'pruned' SSA and the large number of unused ϕ-functions found in minimal SSA. The second variation speeds up the program renaming process by efficiently manipulating the stacks of names used during renaming. Our improvement reduces the number of pushes performed, in addition to more efficiently locating the stacks that should be popped. To convert code in SSA form back into an executable form, the compiler must use an algorithm that replaces ϕ-functions with appropriately-placed copy instructions. The algorithm given by Cytron *et al.* for inserting copies produces incorrect results in some situations; particularly in cases like instruction scheduling, where the compiler may not be able to split 'critical edges', and in the aftermath of optimizations that aggressively rewrite the name space, like some forms of global value numbering.[2] We present a new algorithm for inserting copy instructions to replace ϕ-functions. It fixes the problems that we have encountered with the original copy insertion algorithm. We present experimental results that demonstrate the effectiveness of the first two 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. © **1998 John Wiley & Sons, Ltd.**

key words: compilers; code optimization; data-flow analysis; static single assignment form; lost-copy problem; swap problem

INTRODUCTION

Static Single Assignment (SSA) form is an intermediate representation that compilers use to facilitate program analysis and optimization.[1,3] SSA form can be viewed as a sparse representation for the information contained in classic use-definition and

CCC 0038–0644/98/080859–23$17.50

Received 2 May 1997
Revised and Accepted 27 March 1998

definition-use chains.[4] In many applications, it has become the primary program representation. SSA form has two principal properties that provide an advantage over prior representations.

First, SSA form imposes a strict discipline on the name space used to represent values in the computation. Each reference to a name corresponds to the value produced at precisely one definition point. This is the single assignment property. Second, it identifies the points in the computation where values from different control-flow paths merge. At a merge point, several different SSA names, corresponding 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 the building of data-flow analyses such as definition-use and use-definition chains. Essentially, SSA form encodes the information about definitions and uses into its name space and provides a sparse representation of the information chains. The result is a powerful framework for the analysis and optimization of code in a compiler.[5–8]

The original paper on SSA form presented an algorithm for constructing SSA form from the code for a procedure; this version of SSA is called 'minimal' SSA.[1,3] A subsequent paper presented a more complex and expensive algorithm that produces a smaller version of SSA, called 'pruned' SSA.[9] The SSA constructed by the two algorithms differs in the size of its name space and the number of ϕ-functions that must be inserted.

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*.

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-function at those merge points where the analysis indicates that the value is potentially 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 potentially less expensive than the pruned algorithm. The consequence of algorithmic speed is a larger SSA form.

Our experience with using SSA form in a compiler suggests that neither minimal form nor pruned form is ideal for all purposes. Instead, the compiler writer will want to construct different 'flavors' of SSA, depending on the details of how the SSA form will be used. In our compiler, we routinely translate into SSA form at the beginning of a transformation and back to our simple, linear external representation at its end. We have found that different applications of SSA call for different flavors; at different times, our compiler builds both minimal and pruned SSA, along with a third flavor that we call semi-pruned. The ideas in this paper apply to all three flavors of SSA.

This paper presents three algorithmic improvements to the current art of building and using SSA form. The first creates an SSA form called semi-pruned 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 is an algorithmic fix to a problem that arises when translating SSA form back into executable code. The original algorithm for this translation generates incorrect results when applied to the SSA that can result from certain aggressive code transformations. Our improved algorithm corrects these problems. We have implemented all three improvements in our laboratory compiler. The paper presents measurements that demonstrate the impact of the first two improvements. The third improvement is needed to ensure correctness in the face of aggressive SSA-based program transformations.

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.[10] 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 build SSA form from a straight-line fragment of code is trivial; each time a name gets defined, the compiler invents a new name that 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.

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 Control-Flow Graph (CFG). The definition uses a pseudo function, called a ϕ-function. The arguments of the ϕ-function are the names flowing into the convergence, and the ϕ-function defines a single, new name. Subsequent

Before	After
$x \leftarrow \ldots$	$x_0 \leftarrow \ldots$
$y \leftarrow x + x$	$y_0 \leftarrow x_0 + x_0$
$x \leftarrow x + y$	$x_1 \leftarrow x_0 + y_0$
$z \leftarrow x + y$	$z_0 \leftarrow x_1 + y_0$

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

uses of the original name will be replaced with the new name defined by the ϕ-function. This ensures the second criterion stated earlier: each use refers to exactly one name. To understand the impact of ϕ-functions, consider the code fragment shown in Figure 2. Two different definitions of x reach the use. The construction inserts a ϕ-function 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-functions are needed. At each merge point in the CFG, it must consider, for each value, whether or not to insert a ϕ-function. 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-function. When this happens, the name propagates no further along that path. (Subsequent uses refer to the name defined by the ϕ-function.)

The simplest SSA conversion algorithm would insert a ϕ-function 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, which might be termed 'maximal' SSA, might be huge; it may have many more ϕ-functions than necessary. However, it would conform to the two criteria. This algorithm is inefficient, but it captures the essence of the SSA construction process: decide where to place ϕ-functions, then rewrite the name space. The difference between this algorithm and those that follow is optimization; all the other algorithms produce potentially fewer ϕ-functions and, consequently, smaller name spaces.

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-functions and rename variables. The scheme for ϕ-function placement uses information about dominator relationships in the CFG to determine where ϕ-functions 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-functions builds a *dominator tree* for the CFG and calculates *dominance frontiers* for the nodes in the CFG. 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. Intuitively, the dominance frontier of X is the set of nodes that are one edge beyond the region that X dominates. This is precisely the set of

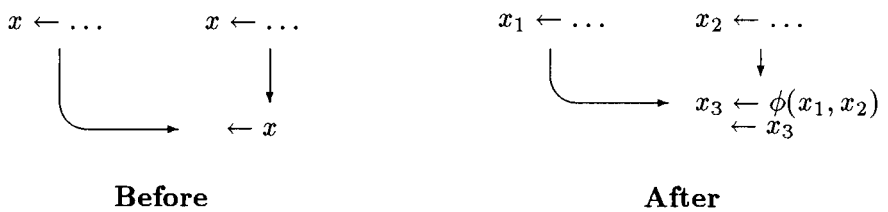

Figure 2. Conversion to SSA form in the presence of control flow

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

    /* STEP 2: Place φ-functions */
    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 3: 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

points where ϕ-functions must be inserted, since it includes only blocks that can be reached along different control-flow paths. To improve the efficiency of ϕ-function placement, both Cytron and Ferrante[11] and Sreedhar and Gao[12] have proposed more efficient schemes. The improvements that we propose in the following sections are also effective in these frameworks.

After ϕ-functions 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-function 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-function parameter in the successor corresponds to the current block.) To continue the search, it recurs 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.

Building pruned SSA

Minimal SSA form relies entirely on dominator information to determine where to insert ϕ-functions. 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-function for *v* at a join point where *v* is not live (see Figure 5).

Choi *et al*. proposed another variation on SSA that they called *pruned SSA*.[9] To build pruned SSA, the compiler first performs 'live analysis' on the routine to define the set of values that are live on entry to the block–that is, values that may be referenced at some later point.[13] Many algorithms exist for computing live information.[4]

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-functions are inserted. The minimal SSA construction inserts a ϕ-function for *v* in *every* node in the iterated dominance frontier of the set of blocks containing an assignment to *v* (denoted $DF^+(\mathcal{A}(v))$). The pruned SSA construction changes this to insert a ϕ-function for *v* in every node $n \in DF^+(\mathcal{A}(v))$, where $v \in \text{LIVE\_IN}(n)$. These changes can drastically reduce the number of ϕ-functions.

The pruned-SSA construction algorithm may cost more than the minimal SSA construction. Not only does inserting ϕ-functions 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[14–16] (and, thus, the asymptotic time complexity of SSA construction 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.* These operations consume a nontrivial amount of time.

* 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.

Equally troubling, building live 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.

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.

USING FEWER ϕ-FUNCTIONS–SEMI-PRUNED FORM

Cytron *et al*. and Choi *et al*. described different flavors of SSA form that vary in the number of ϕ-functions inserted. *Minimal SSA* form places ϕ-functions by looking only at the dominance frontier information without regard to liveness. In other words, it is possible that a ϕ-function will be inserted for a name that is not subsequently used. *Pruned SSA* form relies on live analysis to ensure that no such dead ϕ-functions are inserted. If we are building pruned SSA form, we only insert a ϕ-function 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-functions.

Certain applications require specific flavors of SSA. Cytron *et al*. argued that value numbering would benefit from minimal SSA form–the extra ϕ-functions provide more opportunities to identify congruences.[1] On the other hand, if SSA form is used to support finding live ranges during register allocation, then the pruned form should be used—ϕ-functions represent merging of live ranges, and unnecessary merging detracts from the quality of allocation.[17] Other applications, such as constant propagation[8] and dead code elimination,[4] do not require a specific flavor of SSA. In these applications, extra ϕ-functions do not detract from the quality of analysis; they simply waste space and time.

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-trival 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 resultant ϕ-functions will lie between that of the minimal and pruned forms, but the computation of non-local names is much cheaper to compute than the full-blown live analysis. Therefore, the semi-pruned form represents a compromise between the time required to perform live analysis and the reduction in the number of ϕ-functions 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 for the entire procedure, *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. In our implementations, we use the *SparseSet* data structure, so that the time to perform these actions is constant.[18] The time and space requirements for building *non-local* are, therefore, quite small.

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 bottom of the figure compare the ϕ-functions inserted by the different three flavors of SSA. The minimal SSA form contains ϕ-functions for all three variables. Clearly, the ϕ-functions for *x* and *y* are unnecessary; these variables are never used again. The semi-pruned form does not contain a ϕ-function for *x* because it is not live across any basic-block boundary. However, we still insert a ϕ-function 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-function 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*. show an example where global value numbering may benefit from an extra, dead ϕ-function that minimal would insert.[1] However, these dead ϕ-functions constitute a waste of both time and space for optimizations like constant propagation and dead-code elimination, which can operate on any flavor. Other applications, such as determining live ranges during *register allocation*, depend upon the precision of the pruned form and so must bear the extra cost needed to perform the required live analysis[19,20]

Experimental results

We compared the various flavors of SSA using routines from the SPEC benchmark suite.[21] All tests were run on a two-processor Sparc10 model 512 running at 50 MHz with 1 MB cache and 115 MB of memory. The times represent the average of 10

```
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

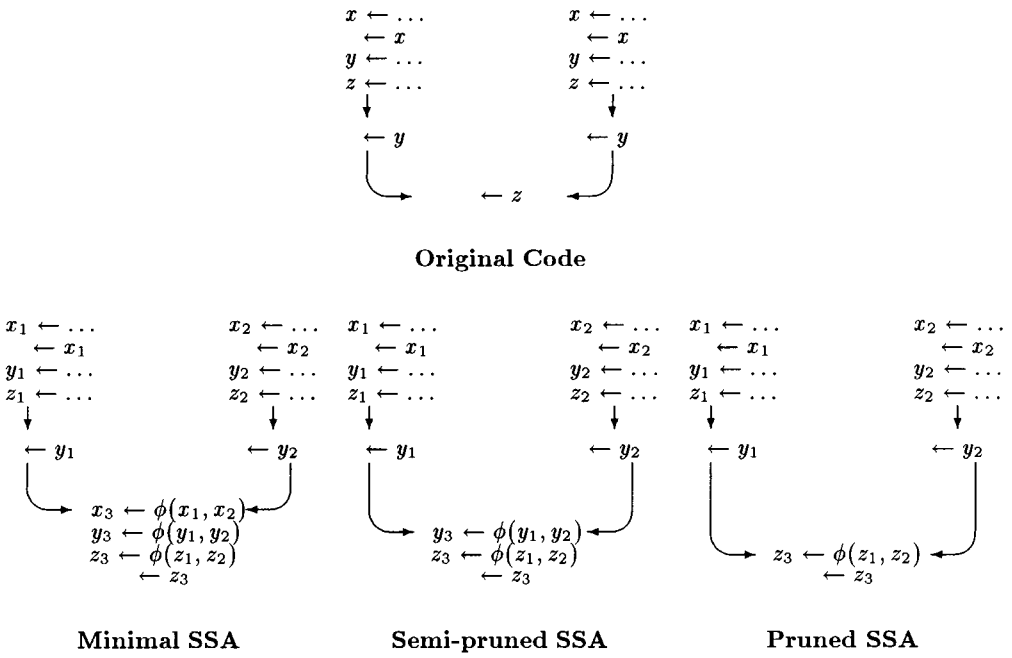

Figure 5. Three flavors of SSA form

runs on a lightly loaded machine. We used iterative data-flow analysis to determine the liveness information required for pruned SSA.[14] Table I shows the number of ϕ-functions and the time required to build the three flavors for each routine (including the time required to perform any necessary data-flow analysis). The number of ϕ-functions required by semi-pruned form always falls between that of minimal and pruned and the time required to build semi-pruned form is almost always 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-functions inserted. The exception is `fpppp`; this routine is composed of one basic block. The semi-pruned and pruned algorithms performed some analysis to determine that no ϕ-functions were needed, but the minimal algorithm proceeded directly to the ϕ-function-insertion step, where no ϕ-functions were inserted.

We also compared the time required for global value numbering (after the code is in SSA form) with each flavor of SSA.[5] The value numbering algorithm requires $\mathbf{O}(E\log N)$ time, where N and E are the number of nodes and edges in the SSA graph.* The experiments confirm that reducing the number of ϕ-functions can improve the execution time of this analysis.

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-functions) defines a

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

Table I. Comparison of three flavors of SSA

Routine	Number of ϕ-nodes			Time to build SSA (sec)			Value numbering (sec)		
	Minimal	Semi	Pruned	Minimal	Semi	Pruned	Minimal	Semi	Pruned
twldrv	73778	11989	9886	1.265	0.332	0.427	8.545	4.890	4.727
deseco	8610	2216	1842	0.231	0.172	0.232	1.972	1.537	1.524
ddeflu	5852	1560	1222	0.116	0.070	0.091	0.803	0.644	0.636
iniset	5364	1080	462	0.127	0.101	0.157	0.343	0.201	0.173
debflu	4715	1748	1542	0.098	0.069	0.087	0.856	0.750	0.738
paroi	3597	767	632	0.079	0.060	0.077	0.352	0.229	0.210
efill	3170	357	74	0.047	0.020	0.025	0.128	0.035	0.030
inisla	2722	267	141	0.048	0.025	0.034	0.134	0.057	0.048
tomcatv	2699	365	145	0.051	0.033	0.042	0.159	0.056	0.060
pastem	2584	374	62	0.055	0.036	0.049	0.183	0.114	0.100
prophy	2021	436	401	0.054	0.042	0.054	0.363	0.296	0.295
inithx	1967	267	85	0.042	0.033	0.044	0.207	0.151	0.148
debico	1880	171	112	0.045	0.030	0.039	0.132	0.075	0.079
repvid	1094	141	45	0.029	0.020	0.031	0.059	0.034	0.029
bilan	1080	70	34	0.028	0.020	0.028	0.097	0.068	0.070
dyeh	857	79	40	0.017	0.011	0.017	0.047	0.030	0.029
sgemm	809	341	279	0.018	0.013	0.013	0.061	0.046	0.037
orgpar	803	143	98	0.027	0.017	0.024	0.080	0.053	0.055
integr	799	89	34	0.016	0.011	0.012	0.040	0.018	0.019
gamgen	761	85	39	0.015	0.008	0.010	0.047	0.022	0.018
heat	667	50	22	0.024	0.018	0.020	0.044	0.030	0.025
fmtgen	653	127	33	0.016	0.007	0.010	0.030	0.017	0.011
inideb	645	148	131	0.015	0.016	0.015	0.065	0.045	0.045
yeh	624	154	122	0.024	0.019	0.024	0.117	0.094	0.090
drepvi	617	76	52	0.024	0.021	0.030	0.060	0.043	0.034
cardeb	601	96	54	0.017	0.013	0.013	0.039	0.021	0.019
ihbtr	597	88	31	0.017	0.011	0.014	0.033	0.019	0.014
bilsla	569	67	38	0.015	0.008	0.011	0.031	0.018	0.013
drigl	557	169	121	0.012	0.010	0.011	0.041	0.030	0.029
saturr	541	27	25	0.024	0.023	0.028	0.074	0.054	0.052
dcoera	334	36	33	0.009	0.007	0.008	0.031	0.022	0.023
sgemv	327	89	61	0.006	0.006	0.008	0.017	0.009	0.012
lissag	311	42	15	0.009	0.006	0.010	0.027	0.017	0.016
colbur	310	15	9	0.014	0.010	0.014	0.034	0.022	0.022
fmtset	275	77	61	0.009	0.009	0.012	0.025	0.019	0.016
sortie	241	19	17	0.009	0.009	0.015	0.027	0.014	0.015
coeray	238	30	26	0.005	0.006	0.004	0.015	0.012	0.014
inter	132	14	7	0.002	0.001	0.006	0.007	0.005	0.003
saxpy	88	12	4	0.004	0.000	0.002	0.003	0.002	0.002
fpppp	0	0	0	0.180	0.220	0.270	0.710	0.760	0.730

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 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-functions 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 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 record which variables have already had a node pushed for the current block. If a variable is redefined inside the block, we 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. The net result is reminiscent of schemes for updating lexically scoped symbol tables on exit from a scope.

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.

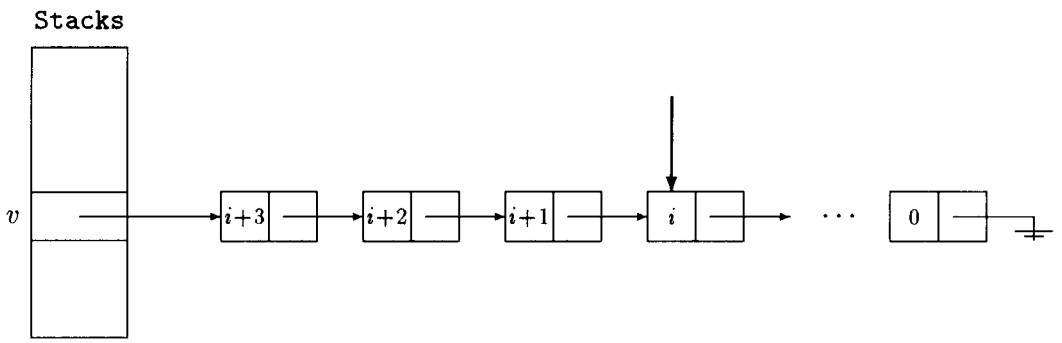

Figure 6. `Stacks` *after variable v is defined three times*

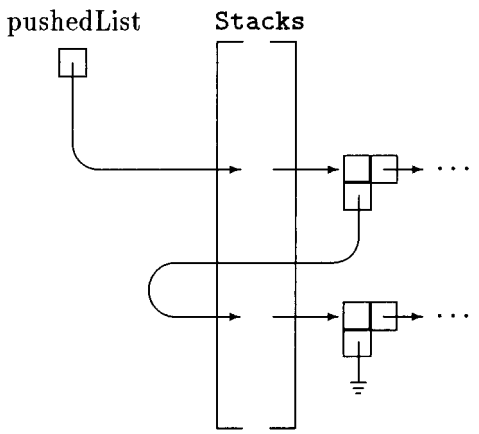

Figure 7. The `Stacks` *data structure, showing the connection between nodes in each stack*

Experimental results

We compared the two stack manipulation methods experimentally. The earlier experimental results section describes the details of the experiment. Table II compares the number of pushes required by each stack manipulation method. 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 compared the total time required to build semi-pruned form using each of the methods. For large routines, a significant proportion of the time is often saved; however, we note that none of these times are very large, so that further exploration in this area would be fruitless. (Note that our implementation uses an Arena-style memory allocator, so allocation costs are minimal.)[22]

REPLACING ϕ-FUNCTIONS WITH COPIES

After optimization, the compiler must translate the SSA form of a routine back into an executable form. We know of no computer that has a hardware ϕ-function; thus, the compiler must translate the semantics of the ϕ-function into commonly implemented instructions. Cytron *et al*. present a simple algorithm for accomplishing this translation.

To replace a ϕ-function in block b, this algorithm inserts a copy operation into each of b's predecessors. Since the meaning of a ϕ-function is a mapping of all the incoming values to a single name, n, placing a copy to n at the end of each predecessor block is equivalent. The copy moves the value corresponding to the appropriate ϕ-function parameter into n. To insert copies for all ϕ-functions, the compiler must iterate through the blocks in the CFG and insert a copy for each parameter of each ϕ-function in the predecessor of the block containing the ϕ-function.

The example in Figure 8 illustrates this process. 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-function. Note

Table II. Comparison of stack handling methods

Routine	Number of pushes		Build SSA (sec)	
	Old method	New method	Old method	New method
twldrv	27295	19569	0.341	0.340
fpppp	19963	5641	0.260	0.222
deseco	14121	8625	0.173	0.171
iniset	6608	5298	0.103	0.096
ddeflu	6393	4651	0.079	0.073
debflu	6389	4225	0.071	0.075
paroi	4881	2864	0.058	0.060
prophy	3609	1947	0.040	0.040
pastem	2755	2060	0.043	0.039
inithx	2686	1706	0.037	0.032
debico	2667	1348	0.031	0.030
tomcatv	2633	1490	0.035	0.033
inisla	2373	1308	0.031	0.026
supp	2037	1734	0.023	0.022
bilan	1994	878	0.024	0.020
subb	1733	1311	0.021	0.020
saturr	1689	1522	0.021	0.023
drepvi	1597	1109	0.023	0.022
yeh	1547	1157	0.023	0.024
orgpar	1499	1053	0.017	0.020
repvid	1449	1010	0.021	0.020
efill	1439	1047	0.018	0.020
inideb	1242	729	0.014	0.016
heat	944	845	0.012	0.014
sgemm	941	681	0.014	0.009
dyeh	941	794	0.014	0.011
cardeb	893	643	0.012	0.012
gamgen	849	417	0.011	0.010
drigl	805	648	0.011	0.011
integr	804	499	0.014	0.009
bilsla	750	339	0.010	0.009
ihbtr	732	605	0.014	0.013
lissag	724	292	0.008	0.007
colbur	716	538	0.010	0.011
fmtset	668	447	0.011	0.009
ftmgen	635	548	0.011	0.013
sortie	595	503	0.010	0.010
dcoera	556	455	0.007	0.006

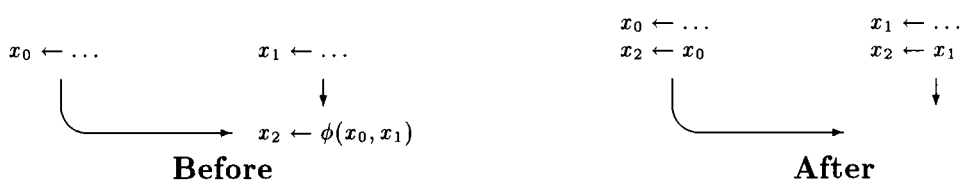

Figure 8. The impact of inserting copies for ϕ-functions

that the insertion of copy operations has made the ϕ-function obsolete, so we can discard it. This translation can insert 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.[19,20]

The algorithm given by Cytron *et al.* works well for the SSA form produced by transformations that do not radically change the name space. Examples include constant propagation[8] and dead code elimination.[4] However, transformations that radically alter the name space can cause the naive copy insertion algorithm to produce incorrect code. For example, aggressive value numbering[5,7] benefits from using SSA form, but can create circumstances that cause the naive algorithm to fail. Copy folding exhibits similar problems. These problems arise from interactions among names along different paths in the CFG. To explain the problem, we first present some necessary background material. Next, we describe two examples that cause problems for the naive algorithm. We use each example to motivate and illustrate part of our new copy insertion algorithm. In the final subsection, we discuss the generality of our new technique.

Background

The problems that break the naive copy insertion algorithm are subtle. They involve both transformations applied to the code while it is in SSA form, and properties of the control-flow graph. This section describes copy folding, a form of subsumption that is particularly well matched to SSA form, and introduces the notion of a critical edge, a feature of some CFGs that can complicate copy insertion.

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, and 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-function. 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. a jump to a location that cannot be determined at compile time), it is not always possible to split critical

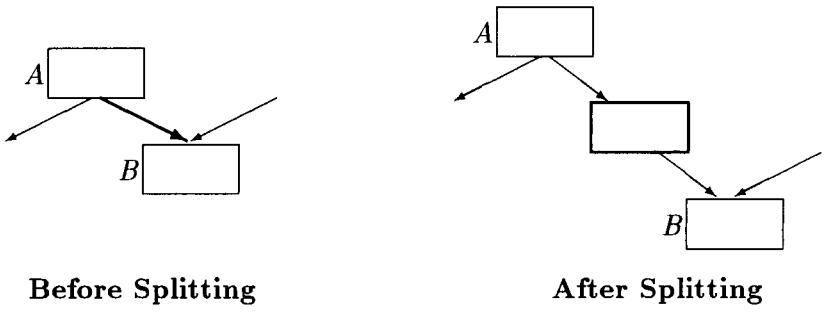

Figure 9. Splitting a critical edge

edges. Similarly, in the late stages of compilation, particularly instruction scheduling, splitting the edge may be both undesirable and impratical. Critical edges are important for code placement algorithms, because their presence can restrict the movement of code and because they can also cause naming conflicts when replacing ϕ-functions with copies.

The 'lost-copy' problem

The *lost-copy problem* can only occur when copies are folded *and* one or more critical back-edges are present. It 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.

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.* The second column shows the code translated into SSA form with copy folding. Notice how y has disappeared. The third column shows the result of replacing the ϕ-function with copies using the naive algorithm. Clearly, the result of the code has changed; it now returns the value of the last iteration. The final column shows how splitting the critical back-edge cures the problem.

Intuitively, the naive copy insertion failed because it created a reference to x_2

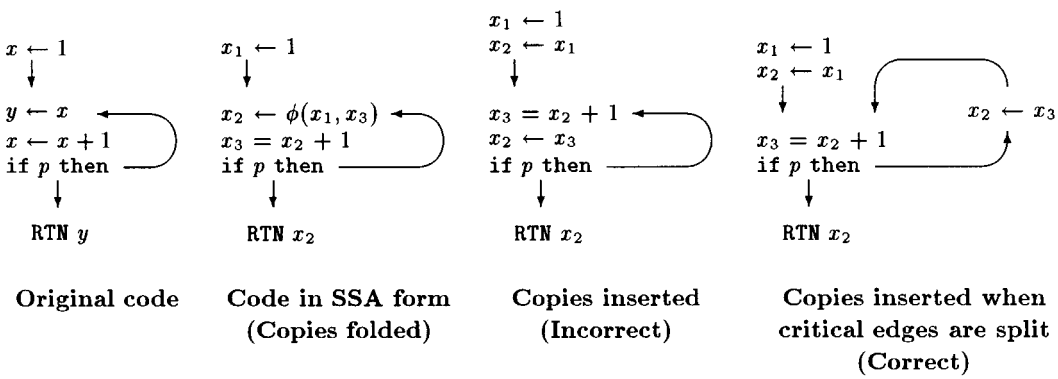

Figure 10. An example of the code leading to the 'lost-copy' problem

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

beyond the scope of the ϕ-function that defined it. 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 name. This is the fundamental idea underlying our copy insertion algorithm. This rewriting 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 needs to push names onto stacks corresponding to the names defined by the inserted copies–these are the only names that need to have their uses rewritten.

The algorithm, shown in Figure 11, 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 replaces uses in ϕ-functions and instructions with any new names. Next, it builds a list of copies that must be inserted and uses the algorithm outlined in the following section to determine the order to insert the copies. If a copy's 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-function that defines the register.

Some careful engineering is required to make this efficient. A block B can be the predecessor to many other blocks, but imagine the case where each of the successor

```
REPLACE_PHI_NODES()
    Perform live 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-function replacement

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-function 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-function, so that it cannot be overwritten. Other solutions exist, but their effect on code size is unpredictable.

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

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-functions.

The right side of Figure 12 shows the result of naively inserting copy operations for the ϕ-functions. 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-functions 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 an ideal solution. It potentially doubles the number of copies necessary for ϕ-function 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-functions are defined by other ϕ-functions in the same block. Notice that the copies inserted into the top block do not contain references to other names defined by a ϕ-function. These copies have

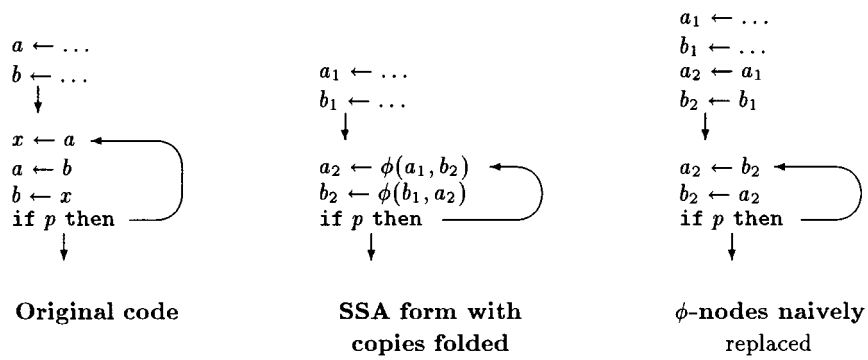

Figure 12. An example of the code leading to the 'swap' problem

been inserted correctly–that is, they do not change the meaning of the code. It is only the copy operations inserted for parameters *that were themselves defined by ϕ-functions in this block* that 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-function in the block. According to the above rule, since the ϕ-function is used as a parameter in another ϕ-function 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-functions 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-functions subject to the following restriction: to schedule a copy c, all other copy operations that include c's destination as their source must be scheduled first. That is, before a name is overwritten, any other operation that needed its value must have it already.

Another way to look at this problem is to model the interaction of the set of copies 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-functions. In the first pass, the compiler counts the number of times a name is used by other ϕ-functions. In the second pass, it builds a worklist of names that are not used in other ϕ-functions. 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

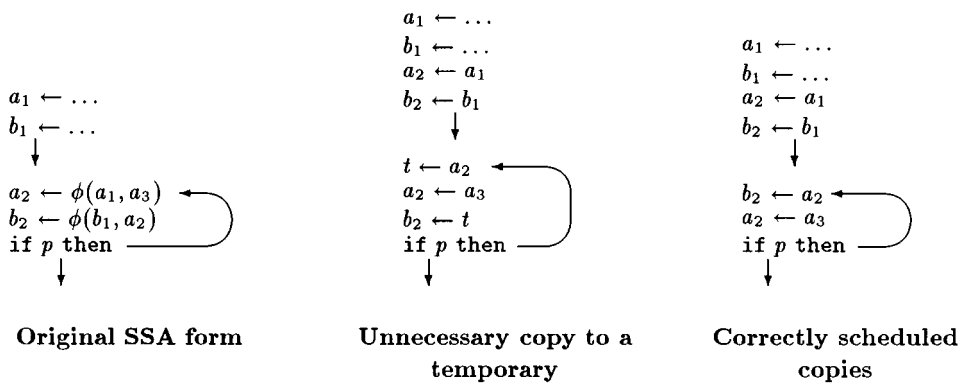

Figure 13. Simple ordering example

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 in which each of the destinations 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 arbitrarily 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-functions 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. This is safe even if this new copy is used as the source for numerous other copies waiting to be inserted. 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 algorithm 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 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 arbitrarily 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 worklist-clearing loop can continue. It alternates between these two sections until all of the copies in the original list have been inserted.

```
schedule_copies(block)
    /* Pass One: Initialize the data structures */
    copy_set ← ∅
    For all successors s of block
        j ← whichPred(s, block)
        For each φ-function dest ← φ(...) in s
            src ← j^th operand of φ-function
            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]
            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

Generality

The preceding subsections developed a new copy insertion algorithm by addressing the problems introduced by two simple and subtle examples. The problem that the new method addresses is actually more general than the examples suggest; the situation can be created by using SSA form with transformations that aggressively move code or rename values.

Abstracting slightly from the example in Figure 12, the lost-copy problem arises when copy folding extends the lifetime of a value x_i past the definition of an argument to the ϕ-function that defines x_i. Specifically, if there exists a use p of x_i such that the path from x_i's definition to p contains a definition of x_j, and x_j is an argument to the ϕ-function defining x_i, then the naive algorithm produces code that delivers x_j to the use at p rather than x_i. We should note that this problem cannot arise inside a single block, because copies are only inserted at the end of a block.

The swap problem is related, but arises due to an important quirk in the definition of SSA form–the assumption that all the ϕ-functions in a block execute concurrently. Parallel execution of the ϕ-functions is important because it allows algorithms that manipulate SSA form to insert ϕ-functions in arbitrary order. If ϕ-function execution were ordered, every algorithm that inserted a ϕ-function would need to perform the same steps that the copy insertion algorithm does. When a set of ϕ-function definitions creates a cyclic flow of values, no serialized order can preserve all the live values, unless an additional temporary name is introduced. This is precisely how the copy insertion algorithm fixes the problem—by introducing a temporary name to break the cycle. Inserting the temporary breaks the cycle and creates an obvious serialization order.

The lost-copy problem is the more general case. It can occur within a block and across blocks. The swap problem is a special case of the lost-copy problem where the path has zero length; it arises as a direct consequence of ϕ-function semantics. The concurrent model of ϕ-function execution simplifies every algorithm that inserts or rewrites ϕ-functions; the price of this simplification must be paid in a more complex copy insertion algorithm.

Our copy insertion algorithm corrects both of these problems. It correctly replaces ϕ-functions with copies in all of the situations that can arise; we know that it cures the problems that we have seen in practice.

CONCLUSIONS

The discovery of SSA form has revolutionized design and implementation of optimizations. 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. All three forms are useful in an optimizing compiler. Choosing between them requires careful consideration of the application, the impact of dead ϕ-functions on the results, and the relationship between SSA construction time and the time required to perform the transformation. 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 live analysis necessary to the pruned form and the large number of dead ϕ-functions found in minimal form. We 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 implementors insights before they begin work.

The second half of the paper tackled the thorny problem of inserting copies for ϕ-functions. We firmly believe that this is a case of Backus' separation of concerns.[23] That is, each optimization pass should not be concerned with its impact on the final transformability of the ϕ-functions, 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-functions 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-functions in control-flow graphs without critical edges. When critical edges are present, we must perform a more complicated algorithm that includes live analysis and a preorder walk over the dominator tree in addition to the existing copy insertion algorithm.

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 sounding board for some of our ideas on ϕ-function-directed copy insertion. The referees have earned our utmost gratitude for their many insightful improvements.

REFERENCES

1. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, 'Efficiently computing static single assignment form and the control dependence graph', *ACM Transactions on Programming Languages and Systems*, **13**(4), 451–490 (1991).
2. L. Taylor Simpson, 'Value-driven redundancy elimination', *PhD Thesis*, Rice University, May 1996.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, 'An efficient method of computing static single assignment form', *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, January 1989, pp. 25–35.
4. K. Kennedy, 'A survey of data flow analysis techniques', in S. S. Muchnick and N. D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.
5. B. Alpern, M. N. Wegman and F. K. Zadeck, 'Detecting equality of variables in programs', *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1988, pp. 1–11.
6. P. Briggs and K. D. Cooper, 'Effective partial redundancy elimination', *SIGPLAN Notices*, **29**(6), 159–170 (1994). (*Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.)
7. P. Briggs, K. D. Cooper and L. Taylor Simpson, 'Value numbering', *Software—Practice and Experience* (1997).
8. M. N. Wegman and F. K. Zadeck, 'Constant propagation with conditional branches', *ACM Transactions on Programming Languages and Systems*, **13**(2), 181–210 (1991).
9. J.-D. Choi, R. Cytron and J. Ferrante, 'Automatic construction of sparse data flow evaluation graphs', *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991, pp. 55–66.

10. K. Kennedy, 'Use-definition chains with applications', *Computer Languages*, **3**, 163–179 (1978).
11. R. K. Cytron and J. Ferrante, 'Efficiently computing ϕ-nodes on-the-fly', *ACM Transactions on Programming Languages and Systems*, **17**(3), 487–506 (1995).
12. V. C. Sreedhar and G. R. Gao, 'A linear time algorithm for placing ϕ-nodes', *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995, pp. 62–73.
13. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
14. J. B. Kam and J. D. Ullman, 'Global data flow analysis and iterative algorithms', *Journal of the ACM*, **23**(1), 158–171 (1976).
15. S. L. Graham and M. Wegman, 'A fast and usually linear algorithm for global flow analysis', *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, Palo Alto, CA, January 1975, pp. 22–34.
16. F. K. Zadeck, 'Incremental data flow analysis in a structured program editor', *SIGPLAN Notices*, **19**(6), 132–143 (1984). (*Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*.)
17. P. Briggs, 'Register allocation via graph coloring', *PhD Thesis*, Rice University, April 1992.
18. P. Briggs and L. Torczon, 'An efficient representation for sparse sets', *ACM Letters on Programming Languages and Systems*, **2**(1–4), 59–69 (1993).
19. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein 'Register allocation via coloring', *Computer Languages*, **6**, 47–57 (1981).
20. P. Briggs, K. D. Cooper and L. Torczon, 'Improvements to graph coloring register allocation', *ACM Transactions on Programming Languages and Systems*, **16**(3), 428–455 (1994).
21. SPEC release 1.2, September 1990. Standards Performance Evaluation Corporation.
22. D. R. Hanson, 'Fast allocation and deallocation of memory based on object lifetimes', *Software—Practice and Experience*, **20**(1), 5–12 (1990).
23. J. Backus, 'The history of Fortran I, II, and III', in Wexelblat (ed.), *History of Programming Languages*, Academic Press, 1981, pp. 25–45.