Incremental Computation of Static Single Assignment Form

Jong-Deok Choi[1] and Vivek Sarkar[2] and Edith Schonberg[3]

[1] IBM T.J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598 (jdchoi@watson.ibm.com)
[2] IBM Software Solutions Division, 555 Bailey Avenue, San Jose, CA 95141 (vivek_sarkar@vnet.ibm.com)
[3] IBM T.J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598 (schnbrg@watson.ibm.com)

Abstract. Static single assignment (SSA) form is an intermediate representation that is well suited for solving many data flow optimization problems. However, since the standard algorithm for building SSA form is exhaustive, maintaining correct SSA form throughout a multi-pass compilation process can be expensive. In this paper, we present incremental algorithms for restoring correct SSA form after program transformations. First, we specify incremental SSA algorithms for insertion and deletion of a use/definition of a variable, and for a large class of updates on intervals. We characterize several cases for which the cost of these algorithms will be proportional to the size of the transformed region and hence potentially much smaller than the cost of the exhaustive algorithm. Secondly, we specify customized SSA-update algorithms for a set of common loop transformations. These algorithms are highly efficient: the cost depends at worst on the size of the transformed code, and in many cases the cost is independent of the loop body size and depends only on the number of loops.

1 Introduction

Static single assignment (SSA) [8, 6] form is a compact intermediate program representation that is well suited to a large number of compiler optimization algorithms, including constant propagation [19], global value numbering [3], and program equivalence detection [21]. In SSA form, each variable appears as a target of an assignment at most once, so that each variable use is reached by a single definition. Special variable definitions called $\phi - functions$ are added to the program to represent multiple reaching definitions.

This paper addresses the problem of *rebuilding* SSA form after program transformation. The traditional algorithm for building minimal SSA form [8, 18] is efficient but exhaustive. Even when program changes are small, the cost of updating SSA with this algorithm is proportional to the size of the procedure. As a result, maintaining current and correct SSA form during a multi-pass compilation process is expensive.

As an alternative to exhaustive SSA reconstruction, we propose incremental SSA update techniques. We present general incremental algorithms for restoring correct SSA form:

– after an arbitrary insertion/deletion of a use/definition of a variable, and
– after an arbitrary update of a single interval.

We also characterize several cases for which the cost of these algorithms will be proportional to the size of the transformed region and hence potentially much smaller than the cost of the exhaustive algorithm.

Additionally, we present customized incremental SSA-update algorithms for a set of common loop transformations, which includes interchange of rectangular/trapezoidal loops, general iteration-reordering loop transformations, and loop fusion. These algorithms are highly efficient: the cost depends at worst on the size of the transformed code, and in many cases the cost is independent of the loop body size and depends only on the number of loops.

It is important to use more efficient customized SSA-update algorithms for common loop transformations because:

– Loop transformations are increasingly important in multi-pass compilers to facilitate instruction scheduling, increase parallelism, improve reference locality, and enable other optimizations [14].
– Loop transformations often constitute a small program change, relative to the size of the procedure.
– SSA form for loops has special properties which make incremental update on a single loop basis efficient and straightforward.

The rest of the paper is organized as follows. Section 2 describes comparison of our method with previous work. Section 3 presents definitions and background material on SSA form. Section 4 presents basic techniques for updating SSA form after changes to individual statements or to individual intervals. Section 5 gives more efficient update algorithms for specific loop transformations, and Section 6 presents our conclusions.

2 Related Work

We focus our discussion on incremental update of SSA form, since it is well-known how to perform incremental updates on other data structures such as the control flow graph and the intermediate language text for these loop transformations. Previous work on incremental data flow analysis focuses on obtaining an updated fixed-point solution after a small program change [5, 4, 13, 12]. These incremental data flow techniques apply to any monotone data flow problem expressed within the classic data flow framework. Our work differs from these exhaustive approaches in that we apply specialized update algorithms for particular changes in the program. Thereby, our methods incur costs proportional to the size of the changes in the program. Cytron and Gershbein [9] also present techniques for incrementally building SSA form to accommodate *may-aliases*.

Their techniques deal with only inserting special definitions, called *may-defs*, generated by may-aliases. They considered incrementality only in response to *specific* demands for data flow information over a program whose structure and contents remain constant. Ryder and Carroll [15] present techniques for incrementally updating the dominator tree of a directed graph.

In [11], Griswold and Notkin propose an incremental paradigm for updating the source abstract syntax tree (AST), control flow graph (CFG), and program dependence graph (PDG) representations of a program so as to properly reflect a program transformation supported by their tool. Each local/compensation transformation that is applied to the source AST by the tool has a corresponding subgraph substitution rule that is applied to the CFG/PDG. In contrast, we present substitution rules for updating the *SSA graph* for a variety of source transformations. Our technique could be used to extend the work reported in [11] so as to allow incremental update of the SSA graph representation in addition to incremental updates of the CFG and PDG. In fact, the SSA graph could be used as a more efficient representation of the data dependence edges in the PDG representation.

In [10], Giegerich et al study the problem of identifying conditions under which data flow information is unchanged after a program transformation is applied. They define the notion of invariance of an approximative semantics with respect to a given set of transformation rules. Their work could potentially be used as a basis for identifying transformations for which the SSA graph remains unchanged.

3 Background

In this section, we describe terminology and definitions used in the rest of the paper.

3.1 Program Representation

Definition 1. A *control flow graph* of a procedure P is a directed multigraph

$$CFG =< N_c, E_c, \textbf{Entry}, \textbf{Exit} >$$

A node $n_i \in N_c$ represents a statement w in P, and an edge $e_k =< n_i, n_j > \in E_c$ represents the transfer of control from the statement of n_i to the statement of n_j. We assume that each node is on a path from **Entry** to **Exit**. □

Definition 2. A node n *dominates* a node $\hat{n}$ in CFG if every directed path from **Entry** to $\hat{n}$ contains n. □

Definition 3. A *back edge* in CFG is an edge $< l, h >$ such that node h dominates node l. Node h is called a *header node.* A back edge defines a *strongly connected region* $STR(h, l)$, which consists of the nodes and edges belonging to all CFG simple paths from h to l, and also the back edge. □

Definition 4. Let $B(h) = \{< l, h > \in E_c \mid < l, h >$ is a back edge$\}$. Then, the *transitive interval with header h*, denoted by $I^*(h)$, is the union of all the $STR(h, l)$ defined by $B(h)$. The *interval with header h*, denoted by $Interval(h)$, is a directed graph $Interval(h) =< N_h, E_h, h >$ such that

- $N_h = \{n \in N_c \mid n \in (I^*(h) - \bigcup_{\hat{h} \in I^*(h), \hat{h} \neq h} I^*(\hat{h}))\}$, and
- $E_h = \{< n_i, n_j > \in E_c \mid n_i, n_j \in N_h\}$.

We denote the header node of interval I by $hdr(I)$. □

Informally, $Interval(h)$ is the union of all the nodes in $I^*(h)$ that do not "belong to" intervals nested in $I^*(h)$, and edges between these nodes.

3.2 Properties of SSA

Rendering a program into SSA form simplifies and increases the accuracy of solving a useful subset of data-flow optimization problems. These results are largely due to the statically functional nature of an SSA form program: each "variable" in the transformed program appears as the target of exactly one assignment statement. All uses are appropriately renamed so that the value flow from a variable's definition (*def*) to its use(s) is explicitly apparent in the program text.

To describe how to construct SSA form of a program, we introduce the following terminology.

Definition 5. Let n and m be nodes in CFG. If n dominates m and $n \neq m$, then n *strictly* dominates m. The *dominance frontier* $DF(n)$ of n is the set of all CFG nodes m such that n dominates a predecessor of m but does not strictly dominate m. □

Note that the dominator relation of a CFG forms a tree, called a *dominator tree*. Algorithmically, SSA form is constructed by identifying regions of a program dominated by a given def. Where control flow from different regions merge, the transformed program contains a ϕ-function that explicitly represents the merge of value defs. More specifically, the set of nodes containing ϕ-functions for variable v can be precomputed as the *iterated dominance frontier* of nodes containing defs of v [8, 18]. The SSA construction algorithm then traverses the dominator tree of the program's control flow graph, maintaining a stack of renamed defs for each variable.

Figure 1 shows an example program segment and its SSA form. Renaming in SSA form ensures that each variable in the transformed program appears as the target of exactly one assignment statement. A ϕ-function is introduced at statement $\hat{S}_4$ to explicitly represent the merge of value defs Y_1 and Y_2. Note that in this paper, we use x and y for the original variable names and use X_i and Y_j for the variable names renamed from x and y, respectively.

A statement in a program has a unique node in the corresponding CFG. Hence, in this paper, we will use statements and nodes (in CFG) interchangeably

S_1:	$x = \ldots;$		$\hat{S}_1$:	$X_1 = \ldots;$	
S_2:	if $(x > 0)$	$y = x + 10;$	$\hat{S}_2$:	if $(X_1 > 0)$	$Y_1 = X_1 + 10;$
S_3:	else	$y = x - 10;$	$\hat{S}_3$:	else	$Y_2 = X_1 - 10;$
S_4:			$\hat{S}_4$:	$Y_3 = \phi(Y_1, Y_2);$	
S_5:	$x = x + y;$		$\hat{S}_5$:	$X_2 = X_1 + Y_3;$	

Fig. 1. An Example program and Its SSA Form

when the meaning is clear. Our results are equally applicable to the case when a CFG node represents a basic block, which could contain multiple statements.

Property 1 Let n be a node in CFG with a def of X_i, and m be a node in CFG with a use of X_i that is a user statement, i.e. *not* a ϕ-function parameter. Then,

1. n dominates m; and
2. there is no node l with a def of x such that l dominates m and is in turn dominated by n.

Definition 6. Given a variable v and user statement S of a procedure in SSA form, $reaching\_def(S, v)$ is the def of v that reaches the statement S. For S_0, which is the entry of the procedure, $reaching\_def(S_0, v)$ is V_0, a def of v assumed to reach the entry of the procedure (interprocedurally).

Since S is a user program statement and not a ϕ*–function*, $reaching\_def(S, v)$ is unique and strictly dominates S. For X_i at a CFG node n to be $reaching\_def(m, x)$, node n and node m must satisfy Property 1.

Property 2 Let n be a node in CFG with a def of X_i, and m be a node in CFG with a use of X_i that *is* a ϕ-function parameter. Then,

1. m is a dominance frontier of n; and
2. $X_i = reaching\_def(S, v)$, where S is a predecessor statement of m in CFG, if S does not have a def of x; or X_i is the def of x at S.

4 Incremental SSA Algorithms

This section explores general incremental properties of SSA. Given an insertion or deletion of a variable def or use, Section 4.1 presents algorithms for restoring correct SSA form. Section 4.2 generalizes these algorithms to work for updates of an interval.

4.1 Incremental SSA Techniques

When a variable def is deleted (inserted), *$\phi-$functions* may have to be deleted (inserted) to restore minimal SSA form.[4] For example, in Figure 2, if the statement $S5$ is deleted in (a), *$\phi-$functions* at statements $S4$, $S7$ and $S9$ must also be deleted, resulting in program (b). Similarly, inserting a def of x at location $\hat{S}5$ in (b) results in the inverse transformation. This insertion update procedure requires using the dominance frontier relation, and all four incremental algorithms presented below use the dominator tree. The variable def that reaches the change site plays a special role.

Theorem 7. Suppose a def of v is deleted (inserted) at statement S in an SSA-correct program P. Let $\hat{P}$ be the resulting SSA-correct program after the deletion (insertion), and $r = \mathit{reaching\_def}(S, v)$ in $\hat{P}$ (P). Then all modified statements in both P and $\hat{P}$ are dominated by r or are in the iterated dominance frontier of r. □

Proof: It follows from Property 1 and Property 2. □

In Figure 2, X_1 is $\mathit{reaching\_def}(S5, x)$ in (b), and dominates all modified statements. (Note that (b) corresponds to the transformed program $\hat{P}$ when S is deleted, and to the initial program P when S is inserted.)

```
S1:  X1 = ...
S2:  if ...
S3:        repeat
S4:              X2 = φ(X1, X3)
S5:              X3 = ...
S6:        until ...
S7:        X4 = φ(X1, X3)
S8:  endif
S9:  X5 = φ(X1, X4)
S10: ... = X5
```

(a)

```
Ŝ1:  X1 = ...
Ŝ2:  if ...
Ŝ3:        repeat
Ŝ4:
Ŝ5:
Ŝ6:        until ...
Ŝ7:
Ŝ8:  endif
Ŝ9:
Ŝ10: ... = X1
```

(b)

Fig. 2. SSA form before and after deleting a def X_3.

A ϕ-function represents the merge of different value defs, which can become *redundant* if one or more of the defs get deleted. For example, the ϕ-functions at $S4$ and $S7$ in Figure 2 represent the merge of defs X_1 and X_3, and will become

[4] By minimal SSA form, we mean the SSA form that would be constructed by the exhaustive algorithm in [8].

redundant if X_3 gets deleted. A formal definition of a redundant ϕ-function is given as follows:

Definition 8. A ϕ-function F is *redundant* if all the parameters, except those identical to the target of the ϕ-function, are the same renamed variable.

Examples of redundant ϕ-functions are:

$$X_i = \phi(X_j, X_j, X_j), \qquad X_k = \phi(X_l, X_k, X_l).$$

In the incremental SSA update algorithms specified below, it is assumed that same-named variables are represented by def-use chains after variable renaming. Specifically, $Uses(d)$ is the set of uses reached by the def d. Using doubly-linked list representations for the $Uses$ sets, adding and deleting uses can be performed in constant time. Computing $r = reaching\_def(S, v)$ can be performed by searching ancestors in the dominator tree, so that the cost is proportional to the height of the tree.

Delete use u of variable v at statement S: Remove u from $Uses(r)$.
Insert use u of variable v at statement S: Add u to $Uses(r)$.
Delete def d of variable v at statement S: Like the original SSA algorithm in [8], the incremental update algorithm works in several steps:

1. *Update Uses sets:* All uses of d, including parameters of ϕ-functions, become uses of r, which is $reaching\_def(S, v)$.
2. *Delete redundant ϕ-functions:* If any ϕ-function F becomes redundant, F will be deleted and the whole process of (a) and (b) repeats, with the target of F as the new d.

Example: To illustrate this algorithm, we delete $S5$ in Figure 2(a). X_3 in $S5$ is used in ϕ-functions at $S4$ and $S7$, and initially $reaching\_def(S5, x) = X_2$. Substituting X_2 for X_3, $S4$ becomes redundant and deleted. Step (a) is applied recursively with X_2 as the newly deleted d. The new $reaching\_def(S4, x)$ is X_1, which is propagated to the ϕ-function at $S7$, where X_2 appears as an input operand after the first iteration. Statement $S7$ is deleted as being redundant, and X_1, as the new $reaching\_def(S7, x)$, substitutes the use of X_4 at $S9$, resulting in a redundant ϕ-function at $S9$. With $X5$ at $S9$ deleted, the use of $X5$ at $S10$ becomes that of $X1$, resulting in Figure 2(b)

Insert def d of variable v at statement S:

1. *Insert new ϕ-functions:* For each node in the dominance frontier of S, if there is no $\phi-function$ for v create a new $\phi-function$, and perform this step recursively. (All the input operands of new $\phi-functions$ are initially assumed to be uses of r. Adjustments are made in step (c), when correct reaching defs are known.) For inserting defs at multiple nodes, a linear time algorithm for computing iterated dominance frontiers of these nodes can be used for placing $\phi-functions$ [18].
2. Update $Uses$ sets for all uses dominated by S, or all uses dominated by any of the new defs of the new $\phi-function$ assignments. This is done by walking down the dominator tree from each of these defs and identifying uses that, along with the def, satisfy Property 1.

3. Update each use that is a parameter of the newly created *$\phi-functions$*, according to Property 2.

Example: To illustrate this insertion algorithm, we insert '$x = \ldots$' at $\hat{S}5$ in Figure 2(b) and rename it X_3. $\hat{S}5$ has two dominance frontiers: $\hat{S}4$ and $\hat{S}7$, at which we insert new *$\phi-functions$*. We then rename the new def of x at $\hat{S}4$ as X_2, and the one at $\hat{S}7$ as X_4, which recursively creates a new def of x at its dominance frontier $\hat{S}9$ as X_5.
We now update *Uses* sets for these new defs. None of X_2, X_3, or X_4 has uses dominated by them. However, X_5 has a use dominated by it at $\hat{S}10$, which used to be a use of X_1, that now becomes a use of X_5. The resulting SSA form is the same as that in Figure 2(a).

The cost of the insert and delete use algorithms is linear in the height of the dominator tree. The cost of the insert (delete) definition algorithm is linear in the size of the subgraph that is dominated by r ($\hat{r}$) or is in the iterated dominance frontier of r ($\hat{r}$). It is in the worst case as expensive as the exhaustive algorithm, but is often much more efficient.

Our incremental algorithm produces minimal SSA form when the CFG is reducible. For an irreducible CFG, restoring minimal SSA form, after a deletion of a def, requires an additional step of identifying and removing a set of ϕ-functions that have as their parameters only themselves plus a single, identical renamed variable. [7]

4.2 Incremental SSA for Updating an Interval

In this section, we generalize the above algorithms to handle arbitrary update of an interval. If interval I is transformed into interval $\hat{I}$, restoring correct SSA form involves: a) rebuilding SSA form of $\hat{I}$ and b) incrementally updating SSA as needed for intervals nested in $\hat{I}$ and intervals containing $\hat{I}$.

We use *augmented control flow graph* (CFG_{aug}) as the program representation. As compared to the original CFG, CFG_{aug} makes loop (interval) structure evident via *preheader* and *postexit* nodes [1, 16]. These extra nodes also provide convenient locations for summarizing data flow information for the loop. An interval I has a single preheader node, denoted by $prehdr(I)$, and there is an edge from the $prehdr(I)$ to $hdr(I)$, the header node. There is a postexit node for each distinct loop exit target. Also, for each interval node n which may exit the interval, there is an edge from n to the corresponding postexit node. Figure 3 shows an example code segment and the corresponding CFG_{aug}. More details on how to compute CFG_{aug} of a CFG are given in [1, 16].

Incrementally Updating Inner and Outer Intervals We first consider how to update the SSA form of inner untransformed intervals.

Definition 9. A use of variable A at node n is *upwards-exposed* at m if there is a CFG_{aug} path $P_{mn} : m \xrightarrow{*} n$ along which A is not defined. A use in I is *upwards-exposed in the interval* if it is upwards-exposed at $hdr(I)$.

Note that if interval I has no def of A, A_j, the unique def of A reaching $prehdr(I)$, will reach all the uses of A in I: all the uses of A in I will be converted into uses of A_j in the SSA form, which are all upwards-exposed at $hdr(I)$ as well as at $prehdr(I)$. If I has any def of A in it, there will be a def ϕ^A_{hdr} at $hdr(I)$. In this case, A_j is still upwards-exposed at $hdr(I)$ as a parameter of the ϕ^A_{hdr}. The following theorem follows from the above observations.

Theorem 10. If an inner interval I' is nested in a transformed interval, then restoring correct SSA form for I' involves updating upwards-exposed uses in I' only. □

Now we consider the effect of a transformation on the SSA form of outer intervals. If a variable A is defined in I but not in $\hat{I}$, $\phi - function$ ϕ^A_{px} at the postexit node of I becomes redundant and need be deleted. Correct SSA form for A in outer intervals can be restored by the procedure specified in Section 4.1 for deleting the def ϕ^A_{px} at the postexit node of I. Similarly, if A is defined in $\hat{I}$ but not in I, a def (ϕ^A_{px}) need inserted at the postexit node of I, so that the incremental insert-definition procedure in Section 4.1 can be applied.

Interval-based SSA Algorithm Correct SSA form for the new interval $\hat{I}$ is restored by performing the steps below. Following [8], S is a set of stacks of defs reaching the current CFG_{aug} node. For each variable A, $S(A)$ initially contains $reaching\_def(prehdr(I), A)$.

1. Rebuild CFG_{aug} dominator tree for $\hat{I}$.
2. Compute dominance frontier for $\hat{I}$.
3. Update $\phi - functions$ in $\hat{I}$.
4. Rename variables in $\hat{I}$, using an algorithm similar to [8], but without modifying the SSA structure of inner loops except for the upwards-exposed uses of variables in them.
5. Incrementally update SSA for outer intervals, using the algorithms in Section 4.1, for variables defined in $\hat{I}$ and not in I, and vice versa.

In many common cases, the set of variables defined in the interval does not change and hence the SSA form of outer intervals need not be updated. We call this the *conservative change property*. For example, this property is satisfied by the loop transformations discussed in Section 5. For such loop transformations, the cost of updating SSA form incrementally using this algorithm is on average linear in $N_I + UE(I)$, where N_I is the size of interval I, and $UE(I)$ is the number of upwards-exposed uses in intervals nested in I.

5 Efficient SSA Update for Common Loop Transformations

In this section, we discuss many popular loop-oriented program transformations, and present incremental update algorithms that can be even more efficient than the algorithms in Section 4 for these special-case loop transformations.

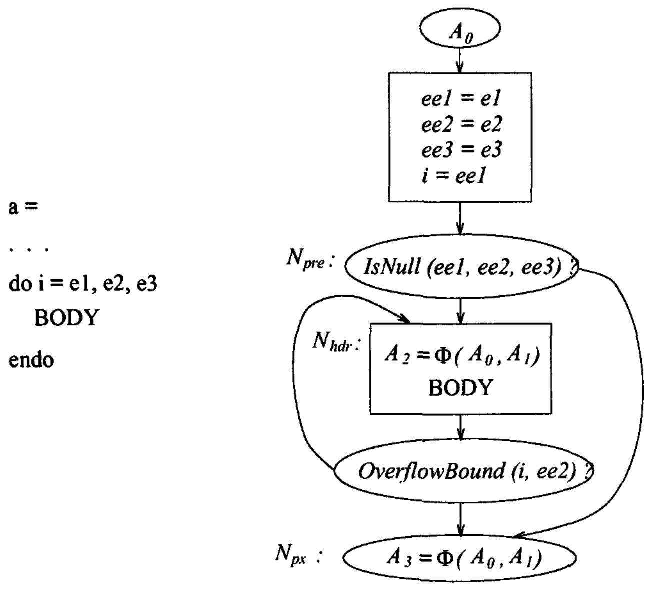

Fig. 3. Schemata of Loop Construct, its CFG_{aug}, and its SSA numbering

5.1 Definition of Loop Construct

Before discussing the various loop transformations, we define the basic loop construct assumed as input to the loop transformations. This loop construct is essentially a well-structured loop with a single entry, single exit, and a single back edge. An example of such a loop construct is a Fortran do-loop with no premature exits. Figure 3 contains schemata of the loop construct, its augmented control flow graph (CFG_{aug}), and SSA numbering for all variables that have at least one def contained within the loop.

We assume that the scope of the index variable (i) is local to a loop construct thus making it unnecessary to perform SSA numbering for loop index variables (since each use of an index variable is associated with a unique loop construct). For convenience, we assume the existence of temporary variables, $ee1$, $ee2$, $ee3$ to capture the values of $e1$, $e2$, $e3$ on loop entry and thus keep those values invariant of the i loop.

We use the name A in Figure 3 to generically represent any variable (other than the index variable) that has at least one def contained within the loop. The four defs of variable A that are noteworthy are: def A_0 reaches the loop header from loop entry, def A_1 reaches the loop header from the back edge (A_0 and A_1 may be user defs or ϕ defs), def $A_2 = \phi(A_0, A_1)$ is the value of A used at the start of the loop body, and def $A_3 = \phi(A_0, A_1)$ is the value of A at loop exit. For all such variables A, there is a ϕ*−function* ϕ^A_{hdr} at the interval header, and a ϕ*−function* ϕ^A_{px} at the postexit node. In this example, A_2 is ϕ^A_{hdr}, and A_3 is ϕ^A_{px}.

5.2 Loop Interchange

The effect of loop interchange [20] is to interchange two perfectly nested loops as shown below in Fortran-like syntax:

```
do i = e1, e2, e3                do j = e7, e8, e9
  do j = e4, e5, e6                do i = e10, e11, e12
    BODY               --->          BODY
  enddo                            enddo
enddo                            enddo
```

The loop body is unchanged by the transformation.

If the loop nest is *rectangular* (i.e. if expressions e4, e5, e6 are independent of variable i), then the bounds expressions are not modified but only relocated ($e7 = e4$, $e8 = e5$, $e9 = e6$, $e10 = e1$, $e11 = e2$, $e12 = e3$). All SSA *Uses* sets remain unchanged after this transformation. The numbering for defs and uses of generic variable A is unchanged by the transformation even though the relative positions of loops i and j have been switched. Since we do not perform SSA numbering for loop index variables, there are no other SSA *Uses* sets that need to be changed. This incremental SSA result is also applicable to any general permutation of n rectangular loops.

In general, we have to consider cases when the loops are triangular or trapezoidal loops (i.e. cases in which e4, e5 are linear functions of i, and e3, e6 are compile-time constants). In such a case, it becomes necessary to generate new loop bound expressions that are different from the old loop bound expressions as in the following example from [20]:

```
do i = 1, n, 1                   do j = 1, n+n, 1
  do j = i, n+i, 1                 do i = max(1,j-n), min(j,n), 1
    BODY               --->          BODY
  enddo                            enddo
enddo                            enddo
```

We observe that a def of a non-index-variable that is used in any of $e7$, $e8$, $e9$, $e10$, $e11$, $e12$ must also have been used in one of $e1$, $e2$, $e3$, $e4$, $e5$, $e6$ (since $e7$, $e8$, $e9$, $e10$, $e11$, $e12$ are derived from $e1$, $e2$, $e3$, $e4$, $e5$, $e6$). Therefore, the algorithm for updating *Uses* sets in trapezoidal loop interchange is equivalent to repeated application of the Insert-use and Delete-use algorithms in Section 4.1), such that uses of non-index-variables in $e1$, $e2$, $e3$, $e4$, $e5$, $e6$ are replaced by uses of the corresponding non-index-variables in $e7$, $e8$, $e9$, $e10$, $e11$, $e12$.

5.3 General Iteration-Reordering Loop Transformation

Any iteration-reordering loop transformation such as interchange, reversal, skewing, unimodular, blocking, coalescing or any combination thereof can be represented by the schema shown in Figure 4 for transforming a set of n perfectly nested loops into a set of n' perfectly nested loops with initialization

```
do $x_1 = l_1, u_1, s_1$
  ⋮
  do $x_n = l_n, u_n, s_n$
    BODY
```

$\Longrightarrow$

```
do $x'_1 = l'_1, u'_1, s'_1$
  ⋮
  do $x'_{n'} = l'_{n'}, u'_{n'}, s'_{n'}$
    $x_1 = f_1(x'_1, \ldots, x'_{n'})$
    ⋮
    $x_n = f_n(x'_1, \ldots, x'_{n'})$
    BODY
```

Fig. 4. General structure of Iteration-reordering Loop Transformations

statements that map the new index variables $x'_1, \ldots, x'_{n'}$ to the old index variables $x_1, \ldots, x_n$ [17]. From our earlier assumption that an index variable is local in scope to its loop construct, we can assume that variables $x_1, \ldots, x_n$ are local in scope to the loop body of the transformed loop nest shown in Figure 4.

As in Section 5.2, we observe that a def of a non-index-variable that is used in the transformed loop nest must also have been used in the original loop nest. Therefore, the algorithm for updating *Uses* sets in a general iteration-reordering transformation is as follows:

1. For each non-index variable A that has at least one def in BODY do
   (a) Let A_0 = def of A that reaches entry to the original loop nest, and A_1 = def of A that reaches end of BODY (as in Figure 3). Also let A_m = def of A on exit from original loop nest. Defs A_0, A_1, A_m will be preserved by the SSA update.
   (b) For each loop $1 \leq i \leq n$ in the original loop nest, delete the ϕ-function associated with A at the entry of loop i (corresponding to ϕ-def A_2 in Figure 3).
   (c) For each loop $2 \leq i \leq n$ in the original loop nest, delete the ϕ-function associated with A at the exit of loop i (corresponding to ϕ-def A_3 in Figure 3). Note that the exit ϕ-function for the outermost loop (A_m) is not deleted.
   (d) For each loop $1 \leq i' \leq n'$ in the transformed loop nest, create a ϕ-function associated with A at the entry of the loop.
   (e) For each loop $2 \leq i' \leq n'$ in the transformed loop nest, create a ϕ-function associated with A at the exit of the loop. ϕ-def A_m will continue to be used as the exit ϕ-function for the outermost loop, $i' = 1$.
2. For each use u' of a non-index-variable in l'_i, u'_i, s'_i s.t. $1 \leq i \leq n'$ do
   (a) Let u = corresponding use in one of l_i, u_i, s_i.
   (b) Insert u' in the *Uses* set that u belongs to (this operation takes constant time with a doubly-linked list representation).
3. For each use u of a non-index-variable in l_i, u_i, s_i s.t. $1 \leq i \leq n$ do
   (a) Delete u from the *Uses* set that it belongs to (this operation takes constant time with a doubly-linked list representation).

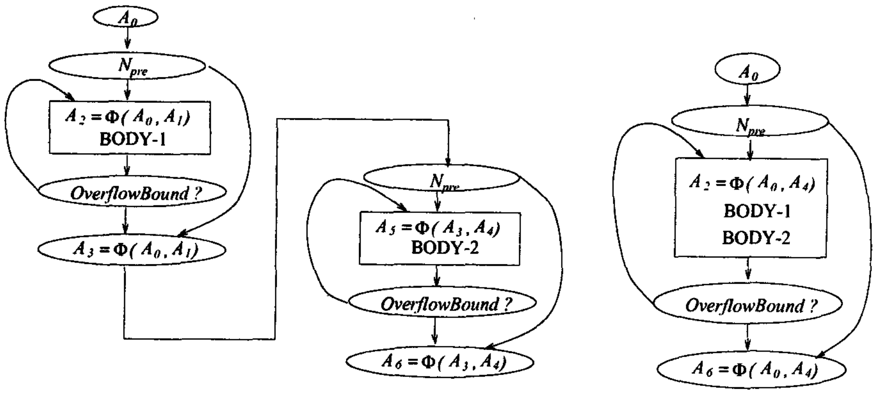

Fig. 5. SSA update for Loop Fusion

These updates will result in *Uses* sets that are identical to the *Uses* sets that would be obtained by exhaustively recomputing SSA form after the transformation. Even though the ϕ-functions associated with a generic variable A are changed, the defs A_0, A_1, A_m remain unchanged after the transformation. The above algorithm ensures that all uses in $l_i, u_i, s_i, l_i', u_i', s_i'$ are properly adjusted. Since we do not perform SSA numbering for loop index variables, there are no other SSA *Uses* sets that need to be changed.

This SSA update has $O(n^2 + n'^2)$ execution-time complexity, which is independent of the size of the loop body.

5.4 Loop Fusion

The effect of loop fusion [2] is to fuse together the bodies of two adjacent conformable loops to obtain a single fused loop body:

```
a =
do i = e1, e2, e3                a =
  BODY-1                         do i = e1, e2, e3
enddo                  --->        BODY-1
do i = e1, e2, e3                  BODY-2
  BODY-2                         enddo
enddo
```

For simplicity, the above schema makes the following assumptions:

1. The two loops together form a single-entry, single-exit region.
2. The two loops use the same index variable and have identical loop bounds expressions.

Figure 5 outlines the low-level control flow and SSA numbering for the loop configurations before and after loop fusion.

The algorithm for updating *Uses* sets after a loop fusion transformation is as follows:

1. Combine $A_2 = \phi(A_0, A_1)$ and $A_5 = \phi(A_3, A_4)$ into a new ϕ-function $A_2 = \phi(A_0, A_4)$
2. Combine $A_3 = \phi(A_0, A_1)$ and $A_6 = \phi(A_3, A_4)$ into a new ϕ-function $A_6 = \phi(A_0, A_4)$.
3. Append $uses(A_5)$ list into $uses(A_1)$ list (this operation takes constant time with a doubly-linked list representation).
4. Delete defs A_3 and A_5.

6 Conclusions

SSA form is a compact intermediate program representation that is well suited to a large number of compiler analysis and optimization algorithms. Though the current SSA construction algorithm has linear execution-time complexity, it is an exhaustive algorithm. For an intermediate form to be practical, it must be efficiently restorable after program transformations. We have therefore examined the question of incrementally maintaining correct SSA form after a number of common program transformations. We have concentrated on program statements and intervals as the basic units of incrementality. By treating intervals as collapsed statements, incremental SSA-updating for intervals can be seen as a generalization of incremental SSA-updating for statements. Finally, we have shown that it is possible to customize SSA-updating for common transformations of structured loops resulting in more efficient algorithms. In some cases (e.g. loop interchange, fusion), the cost is independent of the size of the loop body. In other cases, (e.g. loop distribution) the cost is proportional to the loop body sizes. These sample transformations illustrate that incremental SSA updating is a reasonable approach for a practical compiler: the advantages of SSA form can be repeatedly exploited without multiple costly reconstructions.

References

1. Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the ACM 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, 5(5) pages 617-640.
2. Frances Allen and John Cocke. A catalogue of optimizing transformation. *Design and Optimization of Compilers*, pages 1–30, 1972.
3. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 1–11, January 1988. San Diego, CA.
4. M.G. Burke and B.G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, July 1990.

5. Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.
6. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991.
7. Ron Cytron. private communication.
8. 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*, October 1991.
9. Ron Cytron and Reid Gershbein. Efficiently accommodating may-alias information in ssa form. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1993.
10. Robert Giegerich, Ulrich Moencke, and Reinhard Wilhelm. Invariance of approximative semantics with respect to program transformation. *11th GI Annual Conference, Informatik-Fachberichte 50*, pages 1–10, October 1981.
11. William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.
12. T. J. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. *ACM SIGPLAN Symp. on Principles of Programming Language*, pages 184–196, January 1990.
13. Thomas J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, October 1989.
14. David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.
15. Barbara G. Ryder and Martin D. Carroll. Incrementally updating the dominator tree of a rooted diagraph. Technical report, Rutgers U., December 1986. Center for Computer Aids for Industrial Productivity Technical Report CAIP-TR-029.
16. Vivek Sarkar. The ptran parallel programming system. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.
17. Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, June 1992.
18. Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *22nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 62–73, January 1995.
19. Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, pages 181–210, April 1991.
20. Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989.
21. Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical report, University of Wisconsin, Madison, April 1989. Computer Sciences Technical Report Number 840.