An Efficient Algorithm for the Creation of Single Assignment Forms

Patricia Prather Pineo
Department of Math and Computer Science
Edinboro University of Pennsylvania
Edinboro, PA 16412
ppineo@ediiboro.edu
(814) 732-2405

abstract — Transformation to single assignment form is presented as a technique enabling the exploitation of fine-grain parallelism in programs. An efficient algorithm is presented for the creation of Single Assignment and Static Single Assignment code from unstructured FORTRAN code. The algorithm creates code of near optimal quality with respect to both the number of names and assignment statements added to the code. Experimental results show the degree of enlargement of storage and program length when creating single assignment code, and the containment of enlargement using name reclamation. Other results show the extent of improved parallelization using single assignment code.

1. Introduction

With the advent of parallel and distributed programming environments, it has become increasingly recognized that the reuse of variables imposes unnecessary and costly constraints upon processing. For example, of the three data dependencies, flow dependence, antidependence and output dependence, the latter two arise only when a variable is reused. In addition, the disguised data dependencies known as "side-effects" can only arise when variables are reused.

Disallowing memory reuse dramatically reduces data dependencies and thus exposes available parallelism in an algorithm. When data dependencies are minimized, there is an associated reduction in the need for centralized control, and throughput can increase.

Several parallel programming paradigms have evolved that emphasize the definition and availability of values over the use and reuse of variables. This concept has been explored in a theoretical way by earlier research on dataflow languages and paradigms [Bae97, Del98], and research on functional languages [Koy93].

Freedom from strict sequentiality in the variable space can also be achieved using traditional declarative languages (e.g. FORTRAN) by transforming programs into Single Assignment form. This form each assignment is made into a unique variable, and once computed a variable is never altered. The importance of renaming variables as a programming transformation is growing with the increased recognition of its value in program analysis [Wah96]. Single Assignment has been shown to simplify the problem of program partitioning [Fis91] and is useful in register allocation optimizations [Liu91]. The related Static Single Assignment form has also been demonstrated useful as a pre-processing stage to simplify dataflow analysis during the application of optimizing transformations [Koy93]. In addition, its usefulness has been shown in the elimination of induction variables [Par93].

However, it has been difficult to step beyond the use of renaming as an analysis tool because of the tremendous storage enlargement associated with the form. Earlier work [Fis91, Pine95] describes the Code Liberation technique which devises a general solution to the problem of high-level debugging of parallelized code. Code Liberation creates Single Assignment code for the purpose of tracking non-current variables during debugging. The code can then be parallelized using a parallelizing package of the user’s choice. An unlooked for side-benefit of the technique is a significant increase in parallelism available in the globally renamed code. A second stage of Code Liberation reinspects the code after parallelizing transformations are complete and reclaims the unnecessary introduced names. Reclamation has been shown to successfully reduce the program size to manageable levels prior to execution, and the development of this technique in turn allows the single assignment form to be viewed as practical and useful in additional ways.

1Partially supported by National Science Foundation Grant CCR-91090809 to the University of Pittsburgh.

1060-3425/96 $5.00 © 1996 IEEE
In this paper a practical algorithm is presented for transforming unstructured code to either static single assignment or single assignment form. The algorithm develops renaming techniques for scalar variables in the absence of high level structures (e.g., "dusty deck" FORTRAN code, or intermediate code). The technique is fully extensible to composite variable types. Structured code can also be transformed using the techniques presented here or by recognizing and renaming the structures directly. Details of this second approach and the issues unique to arrays are covered in [Pineo93]. The algorithm presented in this paper is efficient, accessible, and produces code of near optimal quality when considering either the number of lines or the number of names added to the program.

Following the presentation of the algorithm, Name Reclamation is briefly summarized for completeness. Details of the reclamation process and a closer look at the use of Code Liberation for debugging parallelized code are presented in [Pino91, Pino94].

Experimental results are given that show the extent of storage enlargement and program length increase incurred by the creation of single assignment code. Additional results show the degree of improvement of parallelization, and the success of name reclamation.

2. Linear Code Renaming

Work by Cytron, Ferrante, Rosen, Wegman and Zadeck [CRW91] establishes a technique for the creation of Static Single Assignment (SSA) code from unstructured code. SSA code differs from single assignment code in that variable names may be reused if code ranges are reexecuted (as in loops). The algorithm by Cytron et al. is fairly complex (O(R^3), where R=max(N nodes, E edges, A original statements, M original mentions of variables)) but produces code with a minimal number of φ-function insertions. More recent work improves the worst case to "near linear" φ-function insertions. The algorithm performs work on the program in linear time (O(n x \alpha(n))).

In this section a simpler, more efficient (O(n x lvarl)) algorithm is presented that guarantees minimal reassignment insertions when pseudo-loops (see Figure 3) do not occur in the program. In the presence of pseudo-loops the code is semantically correct but not optimal in size. This algorithm is first developed for SSA code, then extended to produce single assignment code.

2.1 Linear Code Renaming to Create SSA Code

Renaming a program for the purpose of creating static single assignment code (SSA) has been approached from a graph theoretic view of the program execution space in the work of Cytron, et al. [CRW91] This approach requires the formation of a control flow graph (CFG), the computation of dominance frontiers, the insertion of φ-functions and finally the renaming of the program. [A dominance frontier of a program block, A, is defined as the set of blocks wherein each member is the first block encountered on some path from A, that is not dominated by A.]

Renaming can be simplified and accomplished more efficiently using the one dimensional view of the lexical code space in preference to the two-dimensional CFG. The program is viewed as a linear sequence of statements where a new name is defined for each variable definition. Variable uses are also renamed by appealing to a list of "current" variable names. In this view each program statement may be tagged with a set of variable names current at that statement, CN(i). This set coincides with the set of definitions reaching that statement.

In programs containing no branches, it can easily be seen that the current name space evolves slowly across statements, and that no variable is ever represented more than once in CN(i). By convention, a variable which has not yet been defined in the program can be represented in CN(i) with a zero extension, x0. Hence the sets CN(i) all have cardinality which equals the number of variables defined in the program or subprogram, lICN\(i\)l = lIvarl.

Figure 1 shows such a program.

<table>
<thead>
<tr>
<th>i</th>
<th>original program</th>
<th>renamed program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y = 2</td>
<td>(a0\ x0\ y1\ y0)</td>
</tr>
<tr>
<td>2</td>
<td>x = y + 3</td>
<td>(a0\ x1\ y1\ y0)</td>
</tr>
<tr>
<td>3</td>
<td>y = x/2-6</td>
<td>(a0\ x1\ y2\ y0)</td>
</tr>
<tr>
<td>4</td>
<td>a = x - y</td>
<td>(a1\ x1\ y2\ y0)</td>
</tr>
<tr>
<td>5</td>
<td>y = a * 3</td>
<td>(a1\ x1\ y3\ x0)</td>
</tr>
<tr>
<td>6</td>
<td>z = x+a+y/2</td>
<td>(a1\ x1\ y3\ z1)</td>
</tr>
</tbody>
</table>

Figure 1. Renamed Program with Current Name Sets

Arbitrary execution order of the linear code may be achieved by inserting branches (and labels) into the code. Each branch from S1 to S2 requires an update from CN(i) to CN(j) to be inserted within the code. This generates the insertion of at most lICN\(i\)l assignment statements of the form \(var = \text{var}_{\text{current at statement}}\) from CN(j), and \(\text{var}_{\text{current at previous statement}}\) is similarly drawn from CN(i). In the case where \(\text{var} = \text{var}_{\text{current at previous statement}}\) is not inserted. The insertions are made at the point immediately preceding the branch.

The method also requires the discovery of loops. Loop heads are places in program execution where code may be entered for the second time. The requirement of SSA code, that each variable have a unique lexical birthpoint in the code, necessitates the creation of a new name at the loop head of each variable defined within the loop. Figure 2 illustrates the transformation. This transformation is similar to the φ-function favored by Cytron, et al.,
presented in Figure 2(c) for comparison. The tag is a method for explicitly tracing the flow graph edge by which a block is entered. The \( \phi \)-function will choose the \( n \)th list item when tag equals \( n \). Cytron, et al., do not introduce tag code or terminology, but such an artifact is necessary if \( \phi \)-functions are actually to be executed.

\[
x = x_1 = x_1
\]

L1: loop

\[
x = x_2 = \phi(x_1, x_4, \text{tag})
\]

L1: loop

\[
x = x_2 = x_2
\]

L2: loop

\[
x = x_5 = x_2
\]

(a) original renamed: (b) without tags (c) with tags

**Figure 2. Renaming a Loop for SSA**

In the transformed code, the loop entry variable, \( x_2 \), is assigned only once each time the loop is entered. This added variable allows the selection of a value from among the definitions reaching the loop head. Significantly, this generates the desirable condition that every program line is reached by a unique name of each variable.

Loops are discovered without a CFG by simply looking for backward branches in the code. All true loops (cycles within a CFG) must have at least one backward branch. This simplified method for identifying loops will discover all loops but may also occasionally uncover "pseudo" loops, where a backward branch is associated with no repeated code. Figure 3 shows such a flow graph. There is little motivation for writing code in this style, which raises the expectation that pseudo-loops occur rarely. In fact, not a single pseudo-loop occurred in ten randomly selected programs included in this experiment. When they do occur, pseudo-loops generate a few unnecessary names in code that is still in valid single assignment form. The additional names may be eliminated by further analysis, or just tolerated considering their infrequency.

The renaming of the program can be accomplished in two passes through the code. The first of these discovers and marks the loop heads (targets of backward branches) and loop tails (locations of backward branches), pseudo-loop heads and their accompanying backward branches, and forward branch points and targets. The second pass then renames the code, line by line, generating and remembering \( CN(i) \) at each statement \( S_i \).

When any loop head is encountered, an entry name assignment for each variable is generated. When the corresponding loop tail is encountered, the statements updating the name space can immediately be inserted just prior to the branch since both \( CN(i) \) and \( CN(j) \) are then known. After the branch the current name of the variable is incremented, creating a loop exit name. (If it is possible to fall out of the bottom of the loop, an explicit assignment to the exit name is also added). The renaming of loops thus generates three sequences of assignment statements, one before the loop to establish loop entry names, one just before the backward branch to catch the "wrap" values, and one after the backward branch to establish exit names. For variables not updated in the loop (the current extension name at loop tail is still the same as that at the loop head), none of these three assignments are needed and they are all deleted. This may cause a name modification within the loop, if such a variable was used in the loop.

When any forward branch is encountered, \( S_i \rightarrow S_j \), a tag \( i \) is attached to label \( j \). When statement \( j \) is reached, a determination is made whether statement \( j \) is a join point, where two or more program paths converge.

If statement \( j \) has only one tag attached to it, and is also preceded by an unconditional branch, it is not a join point. In this case the names of \( CN(i) \) are used in renaming variable uses starting at statement \( j \). Variable definitions continue to generate version numbers beyond the highest numbers yet used. An example of this transformation can be seen in Figure 4(b), statement 3.

If statement \( j \) has at least two entry tags or one tag and the possibility of a "full through" path from above, then statement \( j \) is a join point. In this case, typically update assignments \( var_j = var \) are patched in prior to statement \( j \). In the less usual case that \( var_j \) has a higher version number than \( var \), the reverse assignment is patched in before statement \( j \).

This process has expected time efficiency \( O(N) \) in the length of the original program. In the worst case, however, where each statement is a join point (as in \( n \) nested loops), there could be as many as \( 2n \) \( CN(i) \) assignments added at each. This leads to pathological \( O(N^2) \) performance. This algorithm also requires \( O(N^{n+2}) \) space for name storage. These sets are also useful for
value tracking in transformed programs at debug time, as described in [PiSo91, PiSo94, PiSo93], and so may be viewed as having additional benefits (see also [PiSo94]). However, if program transformations and debugging are not to be done, the number of sets stored can be limited to only those statements associated with branch points or targets.

Figure 4 shows an example program renamed by the two methods for the purpose of comparison. The number of assignments inserted by the Linear Code method can be compared directly with the number of tag assigns in the dominance frontier method. In the absence of pseudo-loops, the number of renaming assignments added is minimal in the Linear Code Renaming method, as in the dominance frontier method. In addition, the size of the generated name space is near minimal. For each variable, the number of names equals the number of assignments plus twice the number of loops involving the variable. (The exit name is a convenience rather than a necessity. In the code of Figure 4(b), x2 could be substituted for x6.) Moreover, the efficiency of the algorithm is \( O(n \cdot \|\text{var}\|) \) instead of \( O(n^2) \).

renamed code

1: \( xl = \)
   if ( ) goto 3
2: \( = x \)
   \( if ( ) \ x1+1 \)
   \( \text{else } \)
3: \( = x \)
   \( x3 = x1+1 \)
   \( \text{else} \)
4: \( = x \)
   \( x5 = x4 \)
   \( \text{else} \)
5: \( = x \)
   \( x7 = x5 \)
6: \( = x \)
   \( x8 = x4 \)

\[\text{CFG with [Dominance Frontiers]}\]

defines 8 tag assigns, 4 \( \phi \)-functions, \( O_n(n^2) \)

Figure 4(a). Dominance Frontier Renaming

renamed code:

1: \( xl = \)
   if ( ) goto 3
2: \( = x \)
   \( if ( ) \ x2 \)
   \( \text{else} \)
3: \( = x \)
   \( x3 = x2 + 1 \)
   \( \text{else} \)
4: \( = x \)
   \( x5 = x3 + 7 \)
   \( \text{else} \)
5: \( = x \)
   \( x7 = x5 \)
6: \( = x \)
   \( x8 = x6 \)

\[\text{CFG with [Linear Code Renamings]}\]

defines 6 names \( O_n(n\|\text{var}\|) \)

(b) — Linear Code Renaming

Figure 4. Comparison of Renaming by the Two Methods
The reduced name space (and the one missing inserted assign) in the new technique results from not requiring a renaming at join points where one edge is entered by falling through the program (e.g., Figure 5b). An additional name saving of this type can be achieved in the above code by eliminating x5 and assigning its values directly into x2. This is a slight deviation from the strict requirement of SSA code as the resultant code may contain multiple assignments to variables matching original program assignments. However, two other significant conditions are maintained. The first is that only one variable version reaches each statement, which is important in dataflow computations. The second is that only one of the multiple assignments will be executed on any forward path through the program. This allows data presence to function as a synchronizing event during parallel execution, much as it might in a dataflow program. In addition, when loops are expanded (see Section 2.2), this code becomes valid single assignment code. It is possible to generate semantically equivalent code, in which a new name is generated at each forward target, generating a unique name at the join (Figure 5c). Such code is equivalent to the implementation of the $\phi$-function (Figure 5d), and does contain multiple assignments, but only to join variables, which are introduced variables.

The process described has been defined for straight line code interspersed with arbitrary branch commands. Structured commands such as the IF THEN ELSE and DO WHILE structures can be modeled using forward and backward branches and handled equivalently (see Figure 6).

The technique just presented to create SSA code can be naturally extended to accomplish the creation of single assignment code. Static Single Assignment (SSA) code requires only that there appear no more than one assignment to any variable name in the code; however such code may be repeatedly executed, as in a loop. Single assignment code requires that no variable once computed ever be altered. Therefore to create single assignment code it is necessary to expand the assignments in all repeated code (loops). Scalars are expanded to arrays, and the assignments are to array elements subscripted by an iteration counter for the loop. In an arbitrary program, loops may be nested, creating irreducible flow graphs.

The algorithm to create SA code from arbitrary code begins as the last by scanning for and marking branch points and targets. With each loop a unique iteration counter looping subscript, LSi, is associated. These LSi’s are initialized at the beginning of the transformed program (=1). Falling into the loop head from above causes reinitialization of the looping subscript. [This is correct because the loop cannot be entered from above more than once unless another loop, with its own subscript, causes the repetition. This "outer" subscript will have been incremented, allowing the inner subscript to be reinitialized.] Branching into the loop does not cause reinitialization of the LS. Exiting the loop generates an increment of the associated LS. Loops that intersect will define a double subscripted variable expansion over the extent of their intersection range. This convention allows for arbitrary loop intersection including both nested and non-nested loops. Figure 7 shows the transformation of a program to single assignment code. The program is renamed exactly as before except that when loops are entered the associated subscript is added to the names being defined. The loop iteration subscripts are also managed according to the description above. Figure 8 gives the algorithm for transforming arbitrary unstructured code to single assignment form.
3. Name Reclamation

Once the single assignment program has been parallelized, partitioned, analyzed or put to some other use, the name reclamation process will be applied. The object of this stage is to recompact the name space by eliminating unnecessary added names prior to execution. Experimental results show the storage enlargement of a single assignment program to be extreme. Some of these additional names have enhanced the useful parallelism in the program and are thus desirable, but the majority are unneeded.

Name reclamation for programs that are to be executed without the debug option can proceed simply based on variable liveness. Given two versions of a variable defined as

\[ X1 = \]
\[ = X1 \]
\[ X2 = \]

we ask whether \( X2 \) can be reclaimed, that is, can it be subsumed by \( X1 \)? If so, then \( X2 \) and all of \( X2 \)'s uses are renamed to \( X1 \). This can be done only if \( X1 \) is no longer live after \( X2 \)'s definition. Since a parallel environment is assumed, the determination of liveness is extended to include parallel processes.

The reclamation process becomes somewhat more complex when it must retain non-current variables that may be requested at debug time as well as those required for correct execution. To determine those that may not be reclaimed, a range of statements in which a variable version must be available to the debugger is determined. Details of how this determination is made are presented in [Pis091, Pine93]. This "Available" range is united with the "Live" range to create a larger Maintenance Range:

\[ MR = AR \cup LR \]

The question of whether \( X2 \) can be reclaimed then is answered by determining whether the maintenance ranges of the two variables are disjoint.

4. System Experimentation and Results

A prototype system has been built for the purpose of enabling experimentation [Pis091, Pine93]. The system is presently designed for FORTRAN 77 code. Consideration of aliasing constructs (COMMON and EQUIVALENCE) has been excluded from the initial implementation. The system is coded in C for the Sun platform. It consists of a global renaming program which creates single assignment code (about 2,000 lines) and a name reclamation program (about 800 lines).
Algorithm Create SA Code via Linear Code Renaming

1. Scan code marking loop heads (assign loop numbers), loop tails (tag with loop head), forward branches, forward targets (tag with forward branch).

Create variable list

2. Rename the code

Current Loop List (CLL) = empty

Generate code LS, = 1 where i = 1.. max loop #

Initialize all current variable names with zero version numbers,

CN(0) = \{v_1, v_a, v_p, v_y, ..., v_n\} = \{0,0,0, ..., 0\} \{used to rename variable uses\}

LASTnames = \{0,0,0, ..., 0\} \{used to generate new variable names\}

where v_y = version number of jth variable at ith line.

For each program line s,

CN(i) = CN(i-1)

if s is a loop head then

ln = loop number of this loop

add LS, to Current Loop List, along with location of loop head and tail

for each variable v_

insert entry name assignments in code,

e.g. var<v_i><subs> = var<v_j><subs> \{defs from LAST, uses from CN\}

update CN(i) with <v_i><subs> = <v_i><subs>

where <subs> will be the (possibly empty) list of subscripts from the current loop list. The subscript of the current loop will be entered as LS, - 1 to allow wrapping of values \{e.g. <subs> = (LS_1,LS_2-1)\}

generate code "LS, = 1"

rename \(s, CN(i), LAST, CLL) \{see following procedure\}

else if s is a loop tail then

h = corresponding loop head

for each variable v_

if v_y = v_y then \{no update of var, in loop\}

remove corresponding entry name assignment prior to \(s,\)

else

insert wrap code assignment

e.g. var<v_i><subs> = var<v_j><subs>

generate "LS, = LS, + 1"

for \(k = \) any loop this branch will exit,

generate "LS, = LS, + 1"

rename \(s, CN(i), LAST, CLL\)

remove LS, from current loop list (CLL)

for each variable var,

if v_y = v_y then \{no update of var, in loop\}

reclaim entry name v_y = v_y - 1

revise use names of var, in loop

revise v_y entry in CN(h) thru CN(i)

insert exit name assignment

e.g. var<v_i><subs> = var<v_j><subs>

update CN(i) with v_y = v_y + 1

else if \(s,\) is forward branch then

rename \(s, CN(i), LAST, CLL\)

else if \(s,\) is forward target then

f = associated forward branch command list

if lengthlist(f) = 1 and \(s,\) is unconditional branch \{not a join point\}

CN(i) = CN(f) \{restore former name space\}

rename \(s, CN(i), LAST, CLL\)

for each variable var,

if v_y < v_y then

insert name space assign prior to \(s,\)

e.g. var<v_i><subs> = var<v_j><subs>

CN(i) = v_y

else if v_y > v_y then

insert name space assign prior to \(s,\)

e.g. var<v_i><subs> = var<v_j><subs>

CN(i) = v_y

rename \(s, CN(i), LAST, CLL\)

else \{all other statements\}

rename \(s, CN(i), LAST, CLL\)

Store CN(i)

end \{for program line\}

end \{rename the code\}

Procedure Rename \(s, CN1,CN2,CLL\)

for each use of var, in \(s,\)

replace var, with var<v_i><subs> using CN1

for each definition of var, in \(s,\)

replace var, with var<v_i><subs> using CN2, CLL

update \(<v_i>+1<subs>\) in CN1 and CN2 with \(<v_i>+1<subs>\)

where \(<subs>\) will be the (possibly empty) list of subscripts from the current loop list

end rename

end \{SA algorithm\}

Figure 8. Algorithm for production of single assignment code
A group of 10 FORTRAN programs was obtained from EISPACK, FFTPACK and LINPACK ranging in size from small to medium (700-800 lines). First the programs were run through global renaming. Parafrase-2 was then applied to the resulting programs. It served as a guide to enable additional parallelization by hand. Name reclamation was then applied. Names were retained whenever two variables were required to be simultaneously live or whenever a value was rendered non-current by transformations applied in parallelization (this policy is in support of the runtime debugging). Thus, somewhat more storage was retained than would be necessary if high-level debugging were not supported. Storage enlargement was computed at both stages.

Table 1 shows the storage expansion measured in the testbed programs as they passed through the stages of the system. Storage is measured in words and is recorded in the original program, after becoming single assignment (stage one), and after parallelization and name reclamation (stage two) have been applied. For example, BAKVEC's original 153 words grows to 11937 in single assignment form, representing an increase of 78 times the original. After parallelization and reclamation, the final storage requirement of BAKVEC is 283 words, or 1.84 times the original.

The degree to which storage is reclaimed varies inversely with the amount of parallelism inherent in the program. Highly parallel programs reclaim fewer names, while programs that underwent no parallelizing transformations had virtually all their introduced names reclaimed. The increases ranged from 1.1 to 7.3 times. The unusually high enlargement figures associated with BQR come from a program with deeply nested loops and several large parallelizable loops.

Table 1. Storage Growth after Global Renaming, parallelization and Name Reclamation

<table>
<thead>
<tr>
<th>program (words)</th>
<th>original</th>
<th>single</th>
<th>after times</th>
<th>name times</th>
<th>storage</th>
<th>increase</th>
<th>reclamation</th>
<th>incr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.BAKVEC</td>
<td>153</td>
<td>11,937</td>
<td>78</td>
<td>283</td>
<td>1.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.BALANC</td>
<td>252</td>
<td>33,401</td>
<td>129</td>
<td>392</td>
<td>1.08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.BALRAK</td>
<td>127</td>
<td>32,721</td>
<td>257</td>
<td>257</td>
<td>2.02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.BANDV</td>
<td>277</td>
<td>735,021</td>
<td>2653</td>
<td>348</td>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.BISECT</td>
<td>150</td>
<td>45,293</td>
<td>300</td>
<td>190</td>
<td>1.27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.BQR</td>
<td>160</td>
<td>4,283,868</td>
<td>26768</td>
<td>1170</td>
<td>7.31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.EZFFTI</td>
<td>39</td>
<td>5,880</td>
<td>150</td>
<td>171</td>
<td>4.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.EZFFTF</td>
<td>6150</td>
<td>23,967,981</td>
<td>3897</td>
<td>7060</td>
<td>1.15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.EZFFTB</td>
<td>6148</td>
<td>28,055,204</td>
<td>4563</td>
<td>13848</td>
<td>2.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.DCHDC</td>
<td>60</td>
<td>14,560</td>
<td>242</td>
<td>164</td>
<td>2.73</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>averages</td>
<td>3903</td>
<td></td>
<td></td>
<td></td>
<td>2.53</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The average storage enlargement measured in these programs was over 3900 times in single assignment form. Excluding the anomalous BQR, the average enlargement was still a discouraging 1368 times. However, after name reclamation the average program size was a more reasonable 2.5 times the original.

Table 2 shows the effect on program length. This figure corresponds roughly with the increase in program complexity introduced by the applied transformations, and also with the time required to compile the programs. Some of the programs included lengthy comment sections, which did not participate in program expansion in any meaningful way. The comments were nonetheless retained through all the program stages. For example, BAKVEC contains 57 comments (93-36) which are also part of the final count of 114 lines after name reclamation. This represents a 1.2 times increase over the original length. Summarizing the counts shows the average program doubled in length in global renaming but was about 1.5 times the original length after name reclamation.

Table 3 shows the increase in parallelizing success experienced using global renaming. The first column shows the number of loops parallelized when the original program was presented to Parafrase-2. The next column registers how many lines of code were in the loop bodies of all those loops combined. This gives a rough measure of the amount of parallelism present in the code. It can be observed that only very small loops (averaging about two statements each) were parallelized when single assignment code was not in use. Studies of the parallelized code revealed that numerous data dependencies often defeated the larger loops.
Table 3. Increase in Parallelism

<table>
<thead>
<tr>
<th>program</th>
<th>without S A</th>
<th>with S A</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loops</td>
<td>loop lines</td>
<td>loops</td>
</tr>
<tr>
<td>1.BAKVEC</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2.BALANC</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3.BALBAK</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4.BANDV</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5.BISECT</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6.BQR</td>
<td>5</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>7.EZFFTI</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8.EZFFTF</td>
<td>24</td>
<td>38</td>
<td>35</td>
</tr>
<tr>
<td>9.EZFFTB</td>
<td>16</td>
<td>58</td>
<td>34</td>
</tr>
<tr>
<td>10.DCHDC</td>
<td>2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>totals</td>
<td>56</td>
<td>119</td>
<td>107</td>
</tr>
</tbody>
</table>

(from totals)

When single assignment code was produced before parallelizing, many more loops (and larger loops) were found parallelizable. This was directly attributable to the reduction of data dependencies. The average loop length increased to over 7 lines, and the number of program lines resident within a parallel loop increased. The final column gives a ratio of column 4 to column 2, showing the increase in parallelized loop lines. A comparison of the total parallelized loop lines in all the programs (760/119) yields the calculated average increase in parallelism, 6.4 times.

5. Related Work

More recent works by Johnson and Pingali and Sreedhar and Gao develop additional alternative graphical approaches for the placing of $\phi$-functions. Each of these algorithms places $\phi$-functions successfully without the computation of dominance frontiers (an O(n^2)) step in Cytron et al.’s algorithm. These algorithms each create a graphical SSA-like representation in linear time, which can be used to generate SSA code in O(EXV). Once $\phi$-functions are placed, the graphs generated by these methods are very similar to Figure 4(a), and will generate equivalent SSA code. This is true even though the means to reach this stage differ substantially in the mentioned algorithms.

The performance orders of these approaches are similar to that of Linear Code Renaming. Close inspection of the Linear Code Renaming algorithm reveals that a “variable size loop” is entered only on loophead, looptail and forward target nodes. These are exactly the nodes that correspond to edges in the CFG.

The real advantage of this technique is that it allows direct transformation of code presented in lexical form, e.g., intermediate code or source level unstructured code, without the requirement of conversion to a graphical representation.

6. Conclusions

The Single Assignment form is increasingly recognized as a valuable form in fitting code to a parallel environment. Its uses include enabling debugging, parallelizing transformations, partitioning code and simplifying dataflow computation.

This paper demonstrates an efficient and practical technique for the computation of Single Assignment code. The experimental results demonstrate that the storage enlargement is controlled efficiently via name reclamation, thus allowing Single Assignment code to be used even more widely.

There is a similarity between a declarative program in Single Assignment form and the same program expressed in a dataflow or functional language. All three forms increase the prospect of parallel execution by passing values through program computations instead of emphasizing variable reuse. The success of these programming paradigms raises the possibility that a dataflow form captures the essence of parallel computation in a more fundamental and natural way than does a traditional declarative approach.

The ability to efficiently create Single Assignment form bridges the gap between these paradigms and allows further experimentation with single assignment principles. Moreover, all three expressive styles suffer from vastly expanded memory requirements, but name reclamation offers a convenient solution when Single Assignment form is used.

Bibliography/References


