A Practical Approach to the Symbolic Debugging of Parallelized Code[1]

Patricia Pineo
ppineo@alleg.edu
(814) 332-2883

Mary Lou Soffa
soffa@cs.pitt.edu
(412) 624-8425

Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
Fax: (412) 624-5299

abstract *-- A practical technique is presented that supports the debugging of parallelized code through global renaming and name reclamation. Global renaming creates single assignment code for programs destined to be parallelized. After parallelization, a reclamation of names not useful for either the execution or debugging of the code is performed. During execution non-current values can then be tracked and reported to the debugger. Results of experimentation indicate the enlargement of the name space is reasonable and that virtually all non-current values are reportable. The technique is independent of the transformations chosen to parallelize the code.*

1. Introduction

The importance of renaming as a program transformation is growing with the increased recognition of its value in program analysis [CyFe87,Wolf89]. The two forms of renaming that have emerged as particularly useful are single assignment and static single assignment. In **single assignment** each assignment is made into a unique variable, and once computed, a variable will never be altered. **Static single assignment** differs in that although only one assignment statement may appear in the code for each variable, that statement can be repeatedly executed (as in a loop).

The usefulness of **static single assignment** has been demonstrated as a pre-processing stage to simplify dataflow analysis during the application of optimizing transformations [CFRWZ91]. It has also been shown useful in applying optimizations such as induction variable elimination [Wolf92]. Under the assumption of **single assignment** code, the problem of partitioning sequential code for a parallel environment is "drastically simplified" [BiNaRo89]. Single assignment is also shown useful in register allocation optimizations [BeDa91].

---

[1] Partially supported by National Science Foundation Grant CCR-91090809 to the University of Pittsburgh.
* Presenting Author

Although these single assignment forms have been shown to be useful for program analysis, their use during program execution has been restricted due to the impracticality of storage enlargement. In this paper we develop a technique that enables the use of renamed code during program execution. This is made possible by selectively reclaiming names prior to code execution.

This technique developed is another application of single assignment code - that of symbolic debugging of code that has been transformed by either traditional optimizations or parallelizing transformations. Because of code modification, deletion, reorganization and parallelization, the actual values of variables seen at breakpoints during runtime will often be different from the values expected by the programmer viewing the sequential, untransformed code. One approach to the problem of non-current variables is to force the programmer to directly view and debug the transformed code, but this approach requires that the user have familiarity with the parallel constructs available, the architecture and the mapping from the source to transformed code. A preferable approach is to allow the user to execute the transformed code on the parallel system but to debug the code from the viewpoint of the sequential code.

This approach to the problem of debugging transformed code has been visited for code transformed for traditional optimizations [Henn82,Zell83,CoMeRu88,PoSo88]. These techniques all create a history of specific optimizations performed with the objective of unwinding the optimizations selectively during debugging in order to recover non-current variables. These techniques work with a subset of 3-4 specific optimizations and must be expanded if other optimizations are applied. They are more successful when optimizations are local, becoming complex and expensive when code is moved across basic blocks. The present work differs in that expansive code motion does not increase the complexity, the work is not transformation dependent and the code is not modified during debugging. This last point is significant because code that is modified for debugging may execute during debugging runs, and then fail when debugging is not invoked.

This problem has also been considered by Gupta[Gupt88] in relation to debugging code reorganized by a trace scheduler. Gupta's technique enables expected values in reordered VLIW code to be reported. It requires debugging requests to be made in advance, and the recompilation of selected traces. The present work differs in that it allows inspection of all variables at any breakpoint without recompilation, and it is not architecture specific.

Each of these methods employs ad hoc techniques for saving and recovering non-current values in newly defined storage locations. By contrast, Global Renaming allows values to be stored and recovered in a unified way, without consideration of any code transformation. Because each value is carried in a unique name, renamed code can be transformed by unrestricted parallelizing transformations, and still be successfully debugged.

Unlike using renaming as a purely analytical technique, renaming in debugging has a problem in the explosion of the storage associated with single assignment programs. This problem is resolved in this work by the application of a second stage that reclaims names not needed for either parallel execution or debugging before execution occurs.

Thus, this paper presents a practical approach to the use of renaming in debugging of parallelized code. The techniques have been implemented and experimental results are presented. Through these experimental results, we demonstrate that after name reclamation, the storage expansion caused by the renaming is reasonable and virtually all non-current names can be reported.

There are several additional advantages of using the renaming approach for debugging transformed code. First, the renaming allows the exploitation of additional parallelism in program code by reducing data dependencies. Further, this same analysis can be used to simplify the application of several standard parallelizing transformations. Finally the technique imposes no restrictions on the number or type of parallelizing transformations applied. This allows the approach to interface easily with a variety of transformational packages aimed at diverse target architectures.

In this extended summary, we first present an overview of the technique. We then present the two analysis techniques, focusing on the reclamation of names. Experimental results are presented, showing that this approach is indeed a practical approach.

2. Overview of Debugging with Global Renaming

Practical high-level debugging with global renaming is accomplished in five stages. An overview of our technique is given in the algorithm of Figure 1. Two stages (numbered one and three) are introduced to bracket the application of parallelizing transformations. The primary purpose of the first stage is the renaming of the code and the production of AVAIL sets, which are sets that retain the current names of variables that should be reportable after the execution of the associated statement number in the original program. These sets provide the value tracking capability used by the debugger at execution time.

Algorithm -- High-level debugging of parallelized code

1. Globally rename code (IN: original code, OUT: single assignment code, AVAIL sets)
2. Apply user chosen parallelization transformations(IN: SA code, OUT:parallelized code)
3. Reclaim unneeded names (IN: parallelized SA code,
OUT: reduced name parallelized code, INOUT: AVAIL)
4. Compile (IN: reduced name parallelized code, OUT: executable code)
5. Execute code through debugger modified to access AVAIL sets when values are requested

Figure 1 -- Overview of the debugging technique

A simple program is shown passing through the stages of the system in Figure 2. Initially the code is globally renamed. This first stage produces a semantically equivalent version of the program in single assignment form, which assigns each (potentially non-current) value a unique storage name. The current names at each statement are retained in the AVAIL sets. The reduction of undesirable data dependencies by the renaming can also be observed in the example. Antidependencies (e.g., statement S1 δ^{-1} S3), and output dependencies (S6 d^{o} S7) are removed in the renamed code. The resulting code has been freed

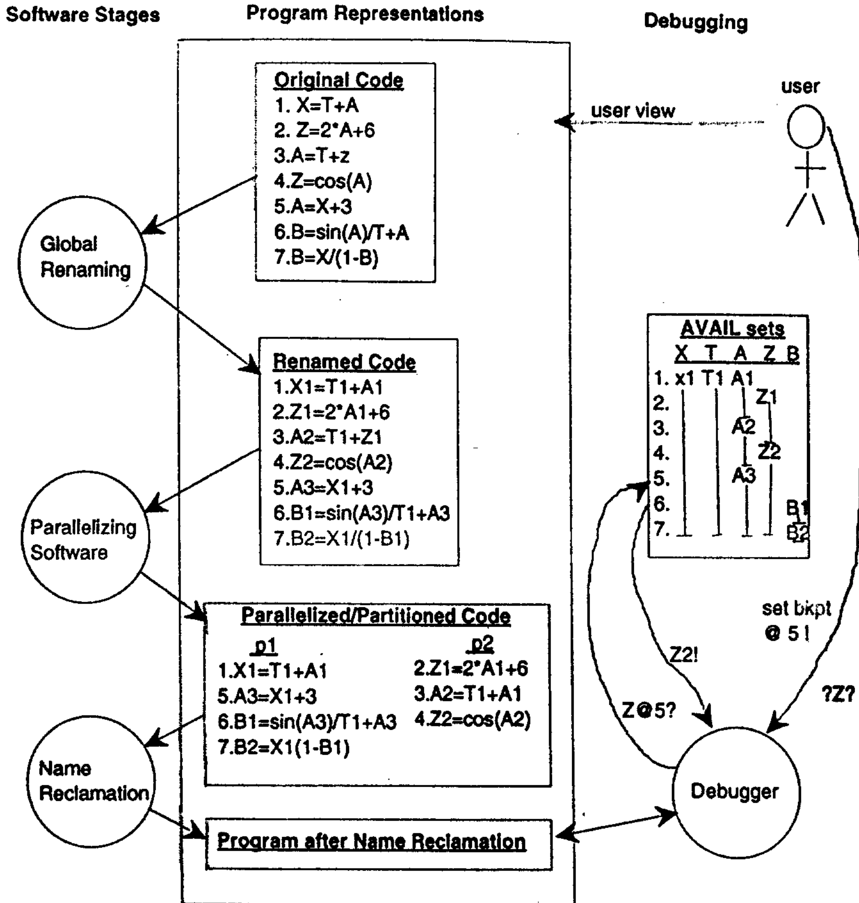


Figure 2 — Debugging with Global Renaming

---

from about half of the original data dependencies and thus allows a more aggressive exploitation of parallelism.

The single assignment code can now be parallelized by software targeted for any desired architecture. The choice of transformations applied in this process are not important to the debugging system. Regardless of where variables are moved, their version names carry the tag required by the debugger for later inquiries.

Once the parallelized code has been finalized, it may be that not all the names introduced through renaming are necessary. Some variables must be retained because they enable the reporting of a non-current value at debug time. In this example, the programmer (debugging from the viewpoint of the sequential code) may insert a breakpoint after statement 5 and request the value of Z. This breakpoint maps to statement of the parallelized code and the associated AVAIL set indicates that Z2 is the proper version of Z to report from the transformed code. Since Z2 must be reported (and not Z1) it is necessary to distinguish between the Zs and therefore the Z2 name must be maintained.

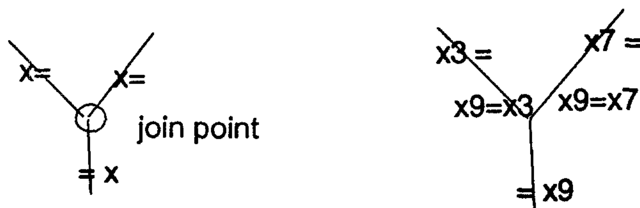
The other reason for not reclaiming names is to allow multiple copies of a variable to be live on different concurrent tasks, thereby enabling the exploitation of parallelism. In this example, A3 cannot share storage with A1, because A1 is simultaneously live on a concurrent process. Similarly A2 cannot share storage with A1 or A3.

The B2 variable is reclaimable because neither B1 nor B2 needs to be available on a concurrent task, nor is B1 live on any concurrent task. The decision to reclaim B2 will result in a change in statement 7 of the parallelized code where B2 becomes B1, and an accompanying update to the database in the B entry of the AVAIL set associated with statement 7.

This parallelized program with names reclaimed (which is no longer single-valued) can now be compiled and executed. The programmer, debugging from the viewpoint of the sequential code, places a breakpoint in the sequential code. This breakpoint maps through to the transformed code. When the breakpoint is encountered, a request for a value made by the programmer traps into the runtime interface. This module in turn replaces the variable name requested with the version name associated with the breakpoint position which is stored in the AVAIL data set. The debugger then proceeds to fill the revised request in the ordinary way. In this example, if the programmer places a breakpoint after statement 3, a request for X, T, A or Z will be replaced with requests for X1, T1, A2, or Z1 respectively and the new requests filled by the debugger. The global renaming and name reclamation processes are presented in greater detail in the following sections.

3. Global Renaming

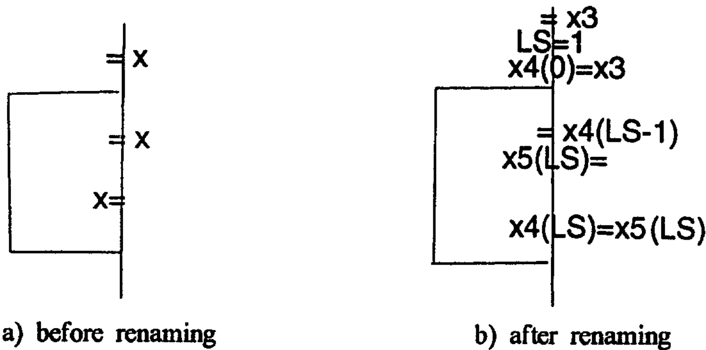
The task of global renaming requires the creation of a new variable name at each variable definition, and also at each program location where divergent execution paths may join. This resolves ambiguity after the join point that may occur in trying to determine which of multiple names (values) should be used. Figure 3 shows this case.



a) before renaming b) after renaming

Figure 3 — Renaming at join points in program flow.

In addition, global renaming must find blocks of code that may be reentered (loops) and ensure that scalars within such blocks are expanded to vectors. This results in variables with altered types as well as altered names. Figure 4 shows this case.



a) before renaming b) after renaming

Figure 4 -- Renaming repeated code

In structured code, these join points and loops coincide with structure boundaries. In unstructured code, they are generally discovered by analysis on a Control Flow Graph (CFG). The variety of these approaches has resulted in the development of three distinct global renaming algorithms. The first is an algorithm for structured FORTRAN 77 code [PiSo91,Pineo93]. This algorithm produces optimal quality code in linear time and recognizes all high-level constructs. It is thus appropriate for use on a large subclass of FORTRAN programs. The second algorithm models unstructured code as a sequence of simple commands in a linear code space, infused with arbitrary GOTO's[Pineo93]. It is able to assert join points without production of the CFG, and so although it is very general, it still operates in linear time. However this algorithm inserts some unnecessary assignment statements. The most general of the algorithms (and the most expensive) is an extension of the dominance frontier algorithm of Cytron, Ferrante, Rosen, Wegman and Zadeck[CFRWZ91] which in its original form produces Static Single Assignment (SSA) code from unstructured code in $O(n^3)$[Pineo93].

The extension necessary to tailor this algorithm to create single assignment code occurs in the discovery and renaming of loops. Each loop discovered in the CFG is assigned a unique looping subscript (analogous to the LS of Figure 4). Individual statements may belong to any number of loops. Variables defined have subscripts added according to loop membership of the statement. The size of the arrays is determined by the loop bounds. If the bounds of the loop are unknown, vectors are allocated as needed in "chunks" of fixed size. In practice these variables are often reclaimed before execution and many of these allocations do not occur.

Although the examples show only scalar variables, array variables are also renamed using analogous techniques. Any time an array is altered it is renamed, initiating a copy into a new array object. The expansion of array objects thus creates arrays of arrays.

While the renaming of arrays continues to remove all anti and output dependencies, it also has the effect of increasing the number of flow dependencies. These come about because the copying of an array object is dependent on the expression defining the new element *as well as* the last current array object. The array assignment A[7]=X is renamed as A2=copy[A1,7,X1]. The renamed code explicitly shows the dependency of the statement on both X1 and A1. The global renaming stage removes these introduced flow dependencies when it can be

determined that they are unnecessary. These approaches and a more detailed discussion of global renaming are described in [Pineo93].

4. Name Reclamation

After the globally renamed program has been partitioned and parallelized, it is the task of name reclamation to eliminate the unnecessary names. This is accomplished in three steps by first computing the maintenance ranges of the values, then reclaiming the unnecessary names, and finally updating the AVAIL sets to reflect the changed names.

4.1 Computing Maintenance Ranges

As seen previously, there are two reasons for maintaining a name: 1) it is still **live**, 2) it still needs to be **available** for debugging. This requires the computation of a maintenance range for each value that includes the entire live range of the value and also its Available range. Symbolically,

$$MR_v = R_{Av} \cup R_{Lv}$$

where R_{Av} is the *available* range of the value computed *in the sequential code* and mapped into the transformed code, and R_{Lv} is the *live* range of the value *in the transformed code.*

It is straightforward to calculate R_{Av} by standard live range analysis with extensions to include statements up through the value redefinition. This is computed by the global renaming stage and stored in the AVAIL data set. However it is then incumbent upon the name reclamation stage to map these availability ranges into the transformed code. In this stage it is necessary to view both the AVAIL sets and the transformed code to determine when specific variables must be available to serve debugging requests in the transformed program. Discrete locales of availability are combined into one contiguous availability range, since variables are assigned only once and can therefore become available only once.

In the computation of R_{Lv}, it is assumed that the transformed program may be modified for some form of parallel execution. A live range for a value may end on a certain processor, but if the value is also live on a parallel task, it cannot be considered dead until there is a synchronization point between the tasks. Therefore live range analysis in a parallel environment requires an inspection of all subtasks that will be in concurrent execution. If a variable is live in only one subtask P1, then the variable dies when the last use is past. However, if the variable is also live on another task P2, then the variable is not dead until the P1-P2 synchronization following the last use. Furthermore, the variable is not completely dead until dead on all subtasks.

To illustrate these computations using the code of Figure 2, the A1 variable is available at statements S1-S2, and must be live at S1-S2. However, since S1 and S2 are on concurrent tasks the live range is extended to the synchronization point. Therefore MR_{A1}=S1-S7. Variable Z1 has an Avail range of S2-S3 and live range of S2-S3, giving MR_{Z1}=S2-S3. For variable Z2 the Avail range, S4-S7, and live range, S4, cross parallel tasks giving a giving a maintenance range MR_{Z2}=S1-S7.

Two variables are said to have overlapping maintenance ranges if they must both be maintained at the same time, as in the case of Z1 and Z2 above. When the two variables have the same root name, eg., X1 and X3, and non-overlapping ranges, it is always safe to reuse the address. Symbolically,

if $MR_{v1} \cap MR_{v2} = \emptyset$
and Root (V_1) = Root (V_2)
then $@V_2 = @V_1$.

The availability of a value can be seen as a further use in generating maintenance ranges. If viewed in this way, the maintenance range within a basic block can be simply defined as beginning at the first position of use of the variable and extending to the last.

In name reclamation, maintenance ranges are computed for each variable in each basic block. These "per block" maintenance ranges are used to create summary maintenance information, such that at each statement it is known whether the maintenance of a particular variable is required *at any time* prior to this statement, or *at any time* beyond this statement. This information is derived from the Control Flow Graph of the program. Backedges are removed from this graph since the reaching definitions of loop variables are handled by explicit mechanisms in renaming. In addition, irreducible flow graph constructs are resolved by removing edges representing backward branches in the written code. The resulting acyclic CFG is used to determine predecessor and successor blocks. Since there may also be concurrent blocks in the CFG, a block X that is concurrent to a block Y is considered both a predecessor and successor to Y.

This graph is then used to create three maintenance sets per block. A Maintenance Range_i set is computed, which holds a minimum and maximum program location for each variable used or available in Basic Block_i. The computation of availability makes use of original statement line numbers that have been appended to the statement during renaming. These numbers indicate the original statement locations of lines of program code. After the application of program transformations, these numbers will normally be unordered and, in addition, may contain duplicated or missing numbers. However, these numbers provide crucial mapping information. Each time a statement line number is encountered, the associated AVAIL set is queried and any variable available at this line has its maintenance range updated with the present program location (in the transformed program).

After the MR_i sets are computed for the blocks, they are used to compute boolean sets, Pre_i and $Post_i$ for each block. Pre_i contains a bit for each variable indicating whether the variable has a maintenance range in any predecessor of BB_i (including concurrent blocks). Pre_i is calculated from the immediate predecessors of BB_i by

$$Pre_1 = \emptyset$$

$$Pre_i = \bigcup_{\substack{j \text{ an imm pred} \\ \text{of } BB_i \text{ or concurrent}}} (Pre_j \cup MR_j)$$

where a non-zero entry in $MR_j.min_k$ defines a true state

$Post_i$ similarly indicates variables that have maintenance ranges in any successor to BB_i. $Post_i$ is calculated in inverse program order from immediate successors and concurrent blocks by

$$Post_{last} = \emptyset$$
$$Post_i = \bigcup_{\substack{j \text{ an imm succ} \\ \text{of } BB_i \text{ or concurrent}}} (Post_j \cup MR_j)$$

4.2 Reclaiming the Names

After the maintenance sets have been computed, names can be reclaimed from the code. The injunction against values sharing a variable name when they have overlapping maintenance ranges allows name reclamation to be modelled as a graph coloring problem. The graph consists of vertices v_i corresponding to each value generated. There is an edge from v_i to v_j whenever v_i and v_j may not share a variable name. Specifically this results when any of the following is true:

1) the variables have different root names,
2) the variables have differing dimensionality, or
3) the variables have intersecting maintenance ranges.

At the beginning of the name reclamation process, this graph contains n vertices and is colored in n colors, where n is the number of variables in the globally renamed program. Name reclamation seeks to *recolor* this graph, using fewer colors. The reclaimed colors represent names that will not appear in the final executable program.

The graph is traversed starting from any arbitrary node. A color pool is maintained which represents the set of names that have been evaluated and will be retained. This set corresponds to the set of names finally held by the *visited* nodes. As the graph is traversed, an attempt is made to recolor each new node encountered with a color already in the color pool. Each candidate color is tried until one is found that has no conflict with the new node, or the list is exhausted. If the node cannot be recolored (the name cannot be reclaimed) then the node's original color is retained and added to the color pool.

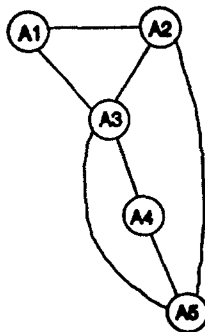


Figure 5 -- Name Reclamation by Recoloring

Figure 5 shows a globally renamed program containing five names, with maintenance range intersections (conflicts) shown as edges. The algorithm starts with an empty color pool and immediately adds A1 to the color set. A2 and A3 are also added because conflicts in the graph do not allow any of these names to share storage. In processing node A4, all colors in the pool (in *last-added* order) will be tested until one is found that does not conflict with A4. If no such color were found, A4 would be retained. However, in this case, after A3 is rejected, A2 is selected to replace A4. In the processing of the A5 node, A3 and A2 are rejected but A1 is selected. The resultant graph contains three names.

The algorithm presented does not compute a minimal name space, as the computation of a minimal name space is an NP-complete problem by a trivial polytransformation from graph coloring. Figure 5 shows that extra names may occasionally be allowed by this algorithm. A4 can be reclaimed by choosing to subsume A4 into either A1 or A2. The choice of A2 as described above will allow A5 to be reclaimed as well (subsumed by A1). However, had the A2 and A1 names been encountered in reverse order, causing A1 to be tried first and chosen, the choice of A1 for A4 forces A5 to be unnecessarily retained. The algorithm tries all active names starting with the last retained and the arbitrariness of this ordering allows nonoptimal name choices to be made in transformed programs. In practice extra names occur infrequently because conflict graphs tend to be characterized by many nodes and few edges.

Computing the maximal degree in the graph allows an upper bound to be placed on the number of colors required = maxdegree + 1. In the graph of Figure 5 the maximum degree is four, and the graph is recolored using three colors. To observe that maxdegree+1 represents an upper bound on retained names in name reclamation, consider the recoloring of the ith node where the degree of $node_i$ <= maxdegree. Assume also that the pool of available colors contains <= maxdegree+1 colors. There are <= maxcolors adjacent to $node_i$. If the color pool contains maxdegree+1 colors, then there exists at least one color not represented on nodes adjacent to $node_i$. This color can be chosen for $node_i$. If the color pool contains < maxdegree+1 colors, then $node_i$'s color can be retained and added to the pool. After the coloring of $node_i$, the color pool still contains <= maxdegree+1 colors.

In untransformed programs, each new definition kills the range behind it and thus there are no maintenance range intersections. As there are therefore no edges, all names are reclaimed except one (i.e., X=>X1).

A criticism of coloring algorithms may be that implementation becomes prohibitively expensive because the graphs involved get quite large. This is especially true for graphs created with single assignment programs. In practice the reclamation algorithm may be implemented without building the graph, using the pre, post and MR sets described above. Collectively they allow the existence of a conflict edge to be efficiently computed.

At each statement the name of a defined variable, V2 may be reclaimed if there exists another variable, V1, previously unreclaimed, such that V1 has the same root name and dimensionality as V2 and the two variables possess nonintersecting maintenance ranges. This last condition is computed by checking

that $MR_{V1} \cap MR_{V2} = \emptyset$ within the block, that V2 has no maintenance range in a predecessor block and that V1 has no maintenance range in a successor block.

If the maintenance ranges are disjoint then the active name replaces the new name and the new name is reclaimed. This also causes maintenance sets for the active variable to be updated. If no active name can be found, the new name is retained and added to the active set.

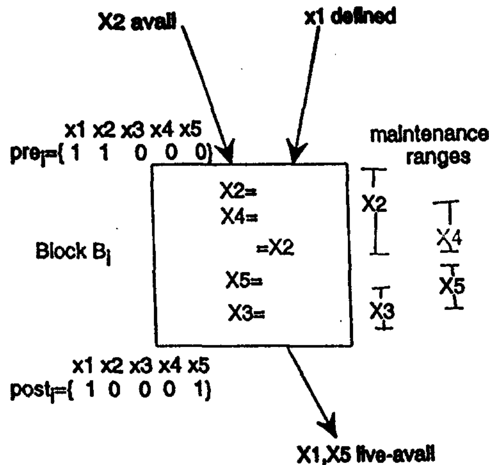


Figure 6 -- Name reclamation in a basic block

Figure 6 illustrates the action of name reclamation. The example is simplified by showing only a single root name. The basic block shown is associated with a pre and post set. These indicate that X1 and X2 have maintenance ranges prior to B and that X1 and X5 have maintenance ranges after B. Beside the block the (contiguous) maintenance ranges are displayed. Active names that reach the block are {X1}. During the processing of the block, X2 will not be reclaimed by X1 because X2 has a previous maintenance range, and also because X1 has a later maintenance range. X2 is then added to the active set. X4 is also retained because it has a nonempty intersection with X2 and X1 has a post maintenance range. X5 will be subsumed by X4 because pre (X5) is false, post (X4) is false and X4 and X5 have an empty intersection. This reclamation causes updates to Pre(X4) and Post(X4) such that

$$Pre(X4) = Pre(X4) \mid Pre(X5)$$
$$Post(X4) = Post(X4) \mid Post(X5).$$

The inblock maintenance ranges of X4 and X5 are also merged and information is retained that X5 is henceforth known as X4 in the *ref-pointer* set. Now X3 cannot be subsumed by X4 because X4 has inherited post(X5). However, X3 can be subsumed by X2, and similar set updates are initiated. The block will finally contain only X2 and X4.

Another form of name reclamation occurs within loops. The algorithm will recover the expansion of objects when the loop in question was not chosen for parallelization. In the case of nested loops, each loop is associated with a

unique looping subscript. Those associated with parallelized loops are retained while the others are reclaimed. The reconstruction of the looping subscript portion of each name is done whenever the name is added to the active set.

At the end of a block's processing, the exiting active set is saved for use by successor blocks. After all blocks are processed, the AVAIL database is updated with the name changes and rewritten for later use by the debugger.

A more detailed view of the name reclamation algorithm is given in the Appendix. The efficiency of this algorithm is bounded by O(plen x |var|) where plen is the length of the transformed program, and |var| is the number of variables in the transformed program.

5. Experimental Results

Global renaming and name reclamation have been implemented in about 3500 lines of C code in a system designed for structured FORTRAN 77. The experimental testbed consists of ten FORTRAN programs taken from the EISPACK and FFTPACK collections.

The issues investigated are:

1) Are there significant numbers of non-current variables in parallelized code?

2) What is the storage increase associated with global renaming, and name reclamation?

3) What factors are responsible for unreclaimed names?

4) What percentage of non-current variables remain unreportable using global renaming and name reclamation?

Table 1 shows the storage expansion measured in the testbed programs as they pass through the stages of the system. Storage is measured in words and is recorded for the original program, after global renaming, and after parallelization and name reclamation have been applied. For the purposes of measuring the storage implied by expanded variables in loops, any loop with uncertain bounds is assumed to execute 10 times[2]. For example, BAKVEC's original 153 memory words grows to 11937 after global renaming, 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 or parallelism inherent in the program. Highly parallel programs reclaim fewer names, while programs that undergo no parallelizing transformations have virtually all their introduced names reclaimed. The increases range 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. The average storage enlargement measured in these programs was over 3900 times after global renaming. Excluding the anomalistic 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.

---

[2] This figure is derived from measurements taken by Knuth [Knut71] who reports the code and execution characteristics of 495 FORTRAN programs.

program	(words) original stor	renamed stor	times incr	after reclamation	times incr
1. BAKVEC	153	11,937	78	283	1.84
2. BALANC	252	33,401	129	392	1.08
3. BALBAK	127	32,721	257	257	2.02
4. BANDV	277	735,021	2653	348	1.25
5. BISECT	150	45,293	300	190	1.27
6. BQR	160	4,283,868	26768	1170	7.31
7. EZFFTI	39	5,880	150	171	4.38
8. EZFFTF	6150	23,967,981	3897	7060	1.15
9. EZFFTB	6148	28,055,204	4563	13848	2.25
10.DCHDC	60	14,560	242	164	2.73
average			3903 (without BQR 1368)		2.53

Table 1 -- Storage Enlargement

In these tests the renamed code was parallelized by Parafrase-2, an automatic parallelizing package licensed through the University of Illinois[PGHLS90]. It was noted that the parallelization of globally renamed code was significantly more successful than when the code was not renamed. Many more (and larger loops) were found parallelizable, an effect that was directly attributable to the reduction of data dependencies. More than six times as many program lines were found in parallelized loops using this technique.

program	names unreclaimed	due to parall	%	due to debugging	%
1. BAKVEC	130	20	15	110	85
2. BALANC	140	120	86	20	14
3. BALBAK	130	120	92	10	8
4. BANDV	71	64	90	7	10
5. BISECT	40	36	90	4	10
6. BQR	1010	1003	99	7	1
7. EZFFTI	132	130	98	2	2
8. EZFFTF	910	644	71	266	29
9. EZFFTB	7700	6506	84	1194	16
10.DCHDC	104	92	88	12	12
average				82	18

Table 2 -- Analysis of Unreclaimed Names

Table 2 shows the analysis of unreclaimed names. Names that are retained because the multiple versions of a variable need to be simultaneously live (as in a parallel loop) were charged to the parallelism column. Conversely, names retained for the purpose of tracking non-current variables were charged to debugging. Where code is reordered aggressively this number of variables charged to debugging can be high (as in the case of BAKVEC), but normally this number is eclipsed by the variables enabling additional parallelism. Over the group of programs, about 82% of the introduced variables enabled parallelization. The remaining 18% were required for value tracking non-current variables.

Program	Total VI	Non-current VI untreated code	%Total	Unreportable using technique	%Total
1. BAKVEC	288	72	25	1	0.3
2. BALANC	2,499	29	1	0	0
3. BALBAK	432	38	9	0	0
4. BANDV	11,160	318	3	1	0.0
5. BISECT	5,565	57	1	0	0
6. BQR	10,332	361	3.5	0	0
7. EZFFTI	1,960	420	21	15	0.8
8. EZFFTF	12,470	1967	16	0	0
9. EZFFTB	12,335	2070	17	0	0
10.DCHDC	4,225	96	2	0	0
averages			9.8%		0.01%

Table 3 -- Variable Unreportability at Debug Time

Table 3 shows the measurement of the degree of non-currentness that exists in the parallelized programs. The number of variable instances was computed as the number of program variables times the number of program lines (only lines past the initial declarations and comments were counted). After transformations were applied the number of non-current variable instances (VI) was counted by counting the number of lines at which each variable is non-current (unreportable at debug time) and summing them over the variable set. The percentage of non-current variable instances was computed and averaged. Finally the number of unreportable variable instances using the proposed debugging technique was counted. These are places where, if a variable value were requested during debugging, the software would report the value is unavailable due to transformations applied. The last column shows this figure as a percentage of the total variable instances.

Some interesting results emerge from these tests. First the ballooning of storage after global renaming is quite large. From a low of 78 times expansion to a high of 26,000 times expansion (average 3800 times), clearly globally renamed code is far too unwieldy to be used directly. The large variation in this expansion depends (exponentially) on the depth of nesting in the program and (linearly) on the program length.

However, name reclamation succeeds in reducing the required storage to a manageable increase of 2.5 times the original. Of these unreclaimed names, a large majority are instrumental in increasing the parallelism in the program. The contribution of these 82% is clearly seen in the improved parallelism figures. The number of lines of code residing in parallelized loops increases an average of 6.4 times.

The experiments show that the existence of non-current variables in parallelized code is a problem. An average of 9.8% of all variable instances are found in non-current ranges in these programs. This figure was unexpectedly high. Without the debugging technique these would be unreportable at debug time. But using these methods only 0.01% of variable instances were still unreportable (due to eliminated code or code moved forward).

These results demonstrate the viability of the method. Not only do they show that the rather invasive nature of parallelizing transformations produces a large percentage of non-current variables, but they also seem to indicate that the cost of debugging such code is small. One could argue that only 18% of the 2.5x storage increase is due to debugging. Since

18% * 1.5x (new unreclaimed names) = .27

it can be concluded that the storage enlargement cost of debugging transformed code is about one quarter of the original storage.

6. Conclusions

As compilers become increasingly autonomous with respect to the restructuring of code, the problem of debugging such transformed code grows in importance. The approach presented in this paper offers significant advantages to the user. It can be used with any transformational package without placing requirements or limitations on the transformations chosen. While the benefits of modular systems design are well-known, this characteristic is particularly useful with parallelizing packages, since the rapid evolution of defined transformations cripples a transformation-dependent approach.

The formation of single assignment code conveys advantages to later stages of code analysis as well. Parallelization is far more successful and all transformations requiring data dependence analysis are simplified. Code partitions are also computed easily. This work suggests that single assignment code captures properties of flow dependence that are so fundamental to the further manipulation of code, especially in a parallel environment, that it is a very appropriate first step to create this form from the input code via global renaming.

Name reclamation makes this a practical and workable approach by removing the unnecessary name allocations. Using this technique, parallelized programs are constructed in modestly expanded spaces, with far more parallelized code. And, most importantly, these programs can be successfully debugged.

APPENDIX -- The Name Reclamation Algorithm

Algorithm Reclaim Names(P:Procedure)

1. Compute Maintenance Sets (MR_i, Pre_i, $Post_i$)
2. Process Program Blocks Reclaiming the names
3. Update AVAIL data set with changed names
end Reclaim Names

Compute Maintenance Sets

1. Create Program Dependence Graph - Basic Blocks, with concurrency indicated.
Mark loop heads and delete backedges. Original statements are marked with original statement numbers.

2. Read AVAIL Set associated with original sequential program.

3. For each Basic Block BB_i do (in any order)
Mark the beginning and end of the maintenance range MR_i of each variable used or defined in BB_i:
For each program line of transformed program
For each USE or DEF var_k
if $MR_i(k)$.min undefined then $MR_i(k)$.min = program location
$MR_i(k)$.max = program location
For each line number read on input {pointing to original statement location}
for each var_k in AVAIL(line number)
if present loc < $MR_i(k)$.min or undefined then
$MR_i(k)$.min = present location
if present loc > $MR_i(k)$.max then
$MR_i(k)$.max = present location

4. Calculate Pre_i for each BB_i in program order. {Pre is a boolean set indicating which variables have maintenance ranges prior to BB_i}

$$Pre_1 = \emptyset$$

$$Pre_i = \bigcup_{\substack{j \text{ an imm pred} \\ \text{of } BB_i \text{ or concurrent}}} (Pre_j \cup MR_j)$$

where a non-zero entry in $MR_i.min_k$ defines a true state

5. Calculate $Post_i$ for each BB_i in inverse program order. {Post indicates which variables have maintenance ranges after BB_i}

$$Post_{last} = \emptyset$$

$$Post_i = \bigcup_{\substack{j \text{ an imm succ} \\ \text{of } BB_i \text{ or concurrent}}} (Post_j \cup MR_j)$$

end Compute Maintenance

Process Program Blocks

```
1. Initialize
   Active_set_i = Ø   {set of active variable names}
   For each variable k, ref_pointer(k)= 0 {ref_pointerpoints to new
        name if variable is renamed}
2. Rename
   For each Basic Block BB_i (in program order)
    2.1 Update maintenance sets
         for all var_k do
              if (j = ref_pointer_k) != 0 {var_j is new name of var_k}
              then Pre_i(j)  = Pre_i(k)  or Pre_i(j)
                   Post_i(j) = Post_i(k) or Post_i(j)
                   MR_i(j).min = minimum(MR_i(k).min,MR_i(j).min)
                   MR_i(j).max = maximum(MR_i(k).max,MR_i(j).max)

    2.2 Compute Active_set_i =        ∪         Active_set_j
                               j=imm pred BB_i
    2.3 Reclaim names in BB_i
         for each USE (var_k )
             if ref_pointer_k = j (>0) then
                  replace var_k with var_j
         for each DEF (var_k ), try to reclaim name(until reclaimed
                  or list exhausted):
             if ref_pointer_k = j (>0) {variable already reclaimed}
              then replace var_k with var_j
             else
              for each var_A in active_set_i with rootname matching var_k
                  determine whether maintenance ranges are disjoint:
                    if not Pre_i(k)      {no previous maintenance range
                           var_k}
                      and MR_i(k).min >= present program location
                      and not Post_i(A) {no later maintenancerange for
                           var_A}
                      and MR_i(A).max <= present program location
                    then {reclaim var_k, replace with var_A}
                       ref_pointer_k = A
                       Post_i(A) = Post_i(k)
                       MR_i(A).max = MR_i(k).max
              if var_k not reclaimed
                  then {add var_k to active_set_i}
                  recompute looping subscript, retaining only those
                     associated with parallelized loops

    2.4 Save active_set_i {to be used by immediate successor blocks}
end Process Blocks
```

Rewrite AVAIL Sets

```
   For each line_i
     For each root name r
       let var_k = AVAIL (i,r)
       if ref_pointer_k = j then AVAIL (i,r) = var_j
   Rewrite AVAIL set to disk
end Rewrite AVAIL sets
```

References

[BeDa91] M. Benitez and J. Davidson, "Code Generation for Streaming: an Access/Execute Mechanism", *4th ASPLOS Conference*, Santa Cruz, April 1991, pp. 132-141.

[BiNaRo89] L. Bic, M. Nagel, and J. Roy, "Automatic Data/Program Partitioning Using the Single Assignment Principle", *Supercomputing 89*, pp. 551-556, Aug 1989.

[CoMeRu88] D. Coutant, S. Meloy and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", *SIGPLAN '88 Conf on Prog Lang Design and Impl*, Atlanta, GA, June 1988, pp. 125-134.

[CyFe87] R. Cytron and J. Ferrante, "What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation", *Proceedings of ACM Conference on Parallel Programming*. pp 19-27. 1987.

[CFRWZ91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Trans on Programming Lang and Systems*, October 1991, pp. 451-490.

[Gupt88] R. Gupta, "Debugging Code Reorganized by a Trace Scheduling Compiler," *Proceedings of Supercomputing 88 Conference*, 1988.

[Henn82] J. Hennessy, "Symbolic Debugging of Optimized Code", *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 3, July 1982. pp. 323-344.

[Knut71] D. Knuth, "An Empirical Study of FORTRAN Programs", *Software Practice and Experience* 1:2, 1971, pp. 105-133.

[PGHLS90] Polychronopoulos, Girkar, Haghighat, Leung, Schouten, "Parafrase-2 User's Newsletter", Center for Supercomputing R&D, University of Illinois, Urbana Illinois. Fall 1990.

[Pineo93] P.P. Pineo, "The High-level Debugging of Parallelized Code using Code Liberation", Ph.D. Thesis, Department of Computer Science, University of Pittsburgh, April 1993.

[PiSo91] P.P. Pineo and M. L. Soffa, "Debugging Parallelized Code using Code Liberation Techniques", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 20-21, 1991, pp. 102-114.

[PoSo88] L. Pollock and M. L. Soffa, "High-Level Debugging with the Aid of an Incremental Optimizer", *Proceedings of the 21st Hawaii Intl Conference on System Sciences*, January 1988.

[Wolf89] M. Wolfe, Optimizing Supercompilers for Supercomputers, MIT Press, 1989.

[Wolf92] M. Wolfe, "Beyond Induction Variables", *SIGPLAN '92 conf on Prog Lang Design and Impl*, San Francisco, CA, pp. 162-174.

[Zell83] P. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs", *Proceedings of the ACM Sigsoft/Sigplan Soft. Eng. Symp on High-Level Debugging*, March 1983, pp 159-171.