Debugging Parallelized Code Using Code Liberation Techniques

Patricia Prather Pineo
Department of Computer Science
Allegheny College
Meadville, Pa 16335

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pa 15260

Abstract A technique is presented that enables the debugging of transformed and parallelized sequential programs from the point of view of the sequential code. Global renaming is applied first to convert the sequential program to single-assignment form. The code is then parallelized using any desired parallelizing transformation. After parallelization is complete, names that are not useful for parallelization or debugging are reclaimed, limiting the storage enlargement usually associated with single-assignment code. Techniques have been applied in a prototype tool designed for FORTRAN 77 code.

1. Introduction

A considerable research effort has been devoted to the development of program transformations that enable the exploitation of parallelism in sequential programs.\(^1,8,11\) One approach to debugging such parallelized programs is to execute and debug the sequential code running on a uniprocessor under the assumption (usually valid) that the transformations do not change the semantics of the program. However, the execution time demands of a sequential execution may make this approach impractical. In addition, the insertion of communication may introduce nondeterminism that is absent in the sequential code.

Another approach is to debug the parallelized version of the code executing on the multiprocessor system. This approach requires the mapping of transformed code back to source code as well as user familiarity with the parallel constructs used. It may be very difficult to map certain transformations back to source code in a way that makes the source code understandable (e.g., renaming). Even when the user interacts with the transformation system in order to direct the transformations, it is desirable to shield the user from the complexities of the parallel code where possible, especially since the code was originally written as sequential code. A third approach to debugging parallelized code, and the approach advocated in this work, is to execute the parallel version of the code on a multiprocessor but have the user debug from the point of view of the sequential code. However, the code transformations increase the difficulty of debugging such parallelized program code from a sequential viewpoint. Because of code modification, reorganization, and parallelization, the actual values of variables seen at a breakpoint during run time will often be different from the values expected by the programmer viewing the sequential code. Such variables are termed non-current. This is a debugging problem also shared by code transformed for traditional compiler optimization.\(^6,7,9,13\)

For example, the original code shown in Figure 1(a) cannot be directly vectorized due to the data dependence on the variable T and the recurrence on the A array in the inner loop. However, application of the transformations scalar expansion, loop splitting, loop
interchange and vectorization produces the equivalent vectorized code shown in Figure 1(b).

(1) \( T = 0 \)
(2) \( \text{do 102 } I = 1,M \)
(3) \( T = T + \text{sqrt}(I) \)
(4) \( \text{do 100 } J = 1,N \)
(5) \( A(I,J) = A(I,J-1)/T \)
(6) 100 continue
(7) 102 continue

(a) Original Code

(1) \( T(0) = 0 \)
(2) \( \text{do 90 } I = 1,M \)
(3) \( T(I) = T(I-1) + \text{sqrt}(I) \)
    90 continue
(4) \( \text{do 102 } J = 1,N \)
(7) 102 continue

(b) Vectorized Code

Figure 1: Syntax changed by parallelizing transformations

In the debugging of such transformed code, the programmer may insert a breakpoint after statement (4) in the original code, and ask for the value of \( T \) or \( I \). How is the debugger to respond to such a request? Clearly the most desirable response is to report values as they would have been in the sequential code. Making the choice to report values expected by the programmer instead of actual run time values is further complicated by the recognition that variables may have changed substantially in type, form and usage in the transformed code. In the example above, because the scalar \( T \) has become an array in the transformed code, the request for "the value of \( T \)" is no longer valid in the run time code.

In this paper, a method is developed for tracking values of non-current variables (as well as those that are current) during parallel execution for the purpose of communicating the expected values to the user at debugging time. The method attempts to retrieve and report the values which would have been current at the breakpoint placed in the sequential code instead of allowing confusing or misleading values found during parallel execution to be reported to the programmer. Parallelizing and optimizing transformations can move, copy, delete or insert code. The debugging technique presented here is able to retrieve and report values made non-current by code moved backward or forward (if not past the breakpoint). It also correctly handles copied code and code insertions. The debugger fails to retrieve values when the code computing the value is moved ahead of the breakpoint or deleted altogether. In such cases the debugger reports that the proper value is unavailable due to code alterations.

The next sections relate this work to earlier approaches and present an overview of the technique. An example is given and the stages of the debugging technique are defined in detail in the following sections. Finally implementation and experimental results are given and the approach is summarized.

2. Background

Earlier work has been done on the similar problem of high level debugging after traditional optimizations have been applied to code. These approaches rely on maintaining a history of transformations applied and attempting to unwind their effects. These methods fail to report values where code is moved forward ahead of the breakpoint, a limitation also shared by the current work.

Each approach is limited to handling a subset of two to five specific transformations. If transformations outside the set are applied, the techniques fail to report values. In some cases these methods also attempt to recompute non-current values from current values.

In contrast, the current technique produces a global renaming such that all values are preserved for later reporting. The global nature of this renaming technique is more suitable for the pervasive transformations typically associated with parallelizing. In addition the current work is not transformation specific, making it capable of interfacing with an arbitrary choice of transformations. This property is especially important if portable code is to be targeted for a variety of different architectures.

The problem of debugging parallelized code has been considered by Gupta in the context of code reorganized by a trace scheduler for VLIW architecture. Gupta’s work requires the programmer to predefine monitoring requests at compile time for specific variables. Selected traces are then recompiled. The current work differs in that any variable may be inspected at any breakpoint without requiring recompilation. In addition, the current work is not architecture specific.

3. Overview

This work approaches value tracking through the production of single assignment code, code in which
A debugging system is developed that uses three stages, two of which occur as preprocessing stages. The original sequential code is input to the first stage of this system and is renamed, producing a sequential program that is single-valued. In this program each variable assignment establishes a unique variable version name related to the original variable name. The first stage also maintains a mapping from each statement number in the original program to the set of variable version names current at that statement. The single-valued sequential code can then be parallelized or optimized by any desired transformation tailored to a given architecture.

A desirable side-effect of the single-valued code is that all pseudo dependencies (i.e., antidependencies and output dependencies) are removed from the data flow, allowing the exploitation of more parallelism and better debugging capabilities. An undesirable side-effect of producing single-valued code is the extreme enlargement of memory required to run such programs and the accompanying increase in the required namespace. To resolve this problem, it is noted that only a small percentage of the new names may actually be useful in the transformed parallel program. There are two reasons why renamed variables are useful:

1) they enable the reporting of a non-current value at some statement at debug time,
2) they enable parallelism to be exploited, resulting in several versions of the same variable that may be live on different processors simultaneously.

Any variable names that are not useful for either of the above reasons need not be retained. In this case, names can be reclaimed, and thus storage need not be allocated for these variable names.

Hence a fully parallelized, single-valued program is input to a second stage, which is a companion stage to global renaming, that reclaims variable names not producing an active benefit. The output of this stage is a parallelized, non-single-valued program. Such a program can be intelligently debugged through a runtime interface, described as stage three of the system. Stage three traps a request for a variable value made to the debugger, replaces it with the variable version name which would have been current at the corresponding point in the sequential code, and then allows the debugger to respond to the modified request. The effect of these stages is to produce code that is liberated from selected undesirable dependencies that interfere with the parallelization or debugging of the program. Hence the technique is termed "Code Liberation". Figure 2 gives an overview design of the system.

4. A Guided Tour

An example serves to illustrate and motivate the transformational stages described in general terms above. Figure 3(a) shows an original program and Figure 3(b) the results of applying global renaming to that program segment. This code is deliberately simple in order to clearly demonstrate the technique. The set of variable ranges beside each code line represents...
a preliminary mapping of those variable versions that must be available (to the debugger) after the execution of that specific statement in the original code. These AVAIL ranges are stored on the external database of Figure 2. It should be noted that no more than one version of each variable is required by the debugger after any statement of the sequential code, and that some version of each variable is always available to the debugger after it has been initialized and is in scope. The reduction of undesirable data dependencies can also be observed in the example.

1) X = T + A
2) Z = 2 * A + 6
3) A = T + Z
4) Z = cos(A)
5) A = X + 3
6) B = sin(A)/T + A
7) B = X/(1 - B)

(a) original code

1) X1 = T1 + A1
2) Z1 = 2 * A1 + 6
3) A2 = T1 + Z1
4) Z2 = cos(A2)
5) A3 = X1 + 3
6) B1 = sin(A3)/T1 + A3
7) B2 = X1/(1 - B1)

(b) single-valued code and AVAIL ranges

Figure 3: Example program

Anti dependencies (as in A's use followed by its definition requiring that statement 1(S1) execute before S3), and output dependencies (two definitions of B requiring that S6 execute prior to S7) are removed in the renamed code. The resulting code has been liberated from about half of the original data dependencies and thus allows a more aggressive exploitation of parallelism.

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

The process of parallelization often includes a phase where code is partitioned for available processors. Under the assumption of single-valued code, the evaluation of alternative partitions can be easily accomplished. Figure 4 shows a partitioned parallelized version of the original program. It can readily be computed that this partition requires no interprocessor communication, since each variable version has a unique birthpoint and there are no uses of variables outside the processor of the birthpoint. For each remote processor requiring the use of a variable version, one synchronization or communication cost is incurred. Counting these occurrences gives a quick and accurate evaluation of the communication costs (assuming uniform distance between processors) implied by a partition under consideration.

Thus the choice to transform sequential code to single-valued code can be shown to solve numerous problems associated with global data flow in a parallel environment.

P1
P2
1) X1 = T1 + A1
2) Z1 = 2 * A1 + 6
5) A3 = X1 + 3
3) A2 = T1 + Z1
6) B1 = sin(A3)/T1 + A3
4) Z2 = cos(A2)
7) B2 = X1/(1 - B1)

Figure 4: parallelized code

Once the parallelized code has been finalized, it may be that not all of the additional names are necessary. As noted earlier 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 5 of the parallelized code (Figure 4) and the associated AVAIL set (Figure 3) 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 given 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 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 on the database. 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.

A more detailed view of the techniques of code liberation, global renaming, name reclamation and the runtime interface is given in the next sections.

5. Techniques to Implement Global Renaming

The techniques developed for Global Renaming deal with data types and program constructs in structured FORTRAN 77 code. Techniques have been developed for types including scalars, parameters (pass-by-reference, pass-by-value, copy-restore) and arrays, and statements including assignment, conditionals, loops and procedure calls. The use of COMMON has yet to be explored.

Scalars in straight line code require a new name whenever a new definition is encountered. When encountering scalar definitions in a conditional, one new name must be generated for each definition, plus one for each variable leaving the conditional to establish a common exit name. Figure 5 shows the liberation of straight line and conditional code.

```
1) if (P.gt.X) then 1) if (Pl .gt.Xl) then
2) X = C + 1 2) X2 = Cl + 1
3) X = X/2 3) X3 = X2/2
   4) else  4) else
   5) X = Y  5) X4 = Y1
       X5 = X3
   6) endif 6) endif
7) B = X 7) B1 = X5

(a) original code (b) single-valued code
```

Figure 5: Renaming conditionals

Scalars defined in loops are renamed by the explicit definition of an array subscripted by a loop iteration variable. The loop iteration variable is a compiler variable and is not required to be single-valued. The transformation requires normalization of the iteration variable and the run time discovery of loop bounds used to allocate memory for the array. Loop normalization leads to a consistent naming approach in the transformed code and also anticipates work that is done later in loop vectorization and processor assignment phases. Dynamic allocation of the required storage is simulated through function calls added to the program code. Scalars with values entering the loop, then redefined within the loop, require special handling. The scalar loop algorithm identifies these variables and generates code that uses the entry value on the first loop iteration, and the appropriate loop exit value on each subsequent iteration. These features are shown in Figure 6.

```
1) do I = 3, 7, 0.5
2)   M = TEST * I
3)   N = cos(M/N)
4)   N = TEST + N
5)   endif

(a) original code

N2(0) = N1 {entry name}
alloc(9)
do LS = 1, 9 {normalization}
1)   I1(LS) = (LS-1)*.5+3 {generate subscript}
2)   M1(LS) = TEST1*I1(LS)
3)   N3(LS) = cos(M1(LS)/N2(LS-1))
4)   N2(LS) = TEST1 + N3(LS) {wrap value}
5)   end do

M2 = M1(9) {exit values}
N4 = N2(9)

(b) single-valued code

Figure 6: Renaming DO loops
```

WHILE loops require incremental allocation of expanded scalar arrays in some cases. First there is an attempt to convert the WHILE loop to an equivalent DO loop. If a variable is discovered within the loop with the qualifications of a loop control variable, (constant increment, initial value and termination condition), then conversion is made. Otherwise the loop is retained as a WHILE loop. Entry and exit values are handled in a way similar to DO loops. Since the virtual arrays of scalars being created in WHILE loops have unknown bounds it is necessary to allocate the arrays in chunks through a procedure call simulating a block form of dynamic allocation (which is not explicitly supported by FORTRAN). This technique contrasts to a method developed formerly which dynamically
allocates a scalar on each iteration, the object of a pointer. While technically single-valued, the ambiguities arising from the pointer assignments in the earlier method lead to code that is still not parallelizable. The method presented here allows more flexible exploitation of parallelism in the WHILE loop code, especially where pipelining approaches are relevant.

1) \( X = 15 \)
2) while \((f(X), \gt 0)\) do
3) \( A = X \)
4) \( X = f(X) \)
5) \( B = X \)
6) \( X = g(X) \)
7) end do

(a) original code

1) \( X1 = 15 \)
\( X2(0) = X1 \) \{entry names\}
\( A2(0) = A1 \)
\( B2(0) = B1 \)
\( LS = 1 \)
alloc(max)
2) while \((f(X2(LS-1)), \gt 0)\) do
3) \( A2(LS) = X2(LS-1) \)
4) \( X3(LS) = f(X2(LS-1)) \)
5) \( B2(LS) = X3(LS) \)
6) \( X2(LS) = g(X3(LS)) \) \{wrap value\}
\( LS = LS+1 \)
   if \((mod(LS, max), eq. 0)\)
     alloc(max)
7) end do
\( A3 = A2(LS-1) \) \{exit values\}
\( X4 = X2(LS-1) \)
\( B3 = B2(LS-1) \)

(b) single-valued code

Figure 7: Renaming WHILE loops

A method has also been developed for renaming scalar parameters that are either pass-by-reference, pass-by-value, or copy-restore. An analysis of subprograms is first performed to determine which pass-by-reference or copy-restore parameters are redefined in procedure bodies. The formal parameter list is expanded with a new name for each of these redefined parameters. The procedure calls are similarly expanded for the actual parameters. This treatment allows additional instances of parallelism to be discovered at procedure call sites, since only expanded parameters will receive output values (see Figure 8). In addition, since no storage location is ever reassigned, the only alias problem (in FORTRAN with COMMON excluded) that persists in this method occurs when the same input parameter is used twice in a parameter list. These cases are rare and result in a warning message to the programmer.

\[
\begin{align*}
\text{main program} & & \text{main program} \\
X & = 3 & X1 & = 3 \\
call ASUB(X) & & call ASUB(X1) \\
call BSUB(X) & & call BSUB(X1,X2) \\
\text{print} (X) & & \text{print} (X2) \\
subroutine ASUB(X) & & subroutine ASUB(X1) \\
Y & = \sqrt{X} & Y1 & = \sqrt{X1} \\
\text{print} (Y) & & \text{print} (Y1) \\
end ASUB & & end ASUB \\
subroutine BSUB(X) & & subroutine BSUB(X1,X4) \\
X & = X + 1 & X2 & = X1 + 1 \\
X & = X / 2 & X3 & = X2 / 2 \\
X & = \tan(X) & X4 & = \tan(X3) \\
end BSUB & & end BSUB \\
\end{align*}
\]

(a) original code

Arrays present special problems owing to the size of storage required by copying. This problem is solved by reclaiming unneeded array instances after renaming and again after parallelization has occurred. Arrays can be redefined by partial or total redefinition. In general a change to any element will cause the entire array to be copied and a new array object created. However when an entire array is redefined, without intervening uses, this can be construed as a total array definition and done on a single array object.

A problem unique to arrays is the implied data dependence inherent in a partial array definition. Consider the following assignment statement:

\[ A(7) = X \]

The assignment above has changed \( A \) and so should cause the creation of a new copy of \( A \), which depends on \( X \) in element 7, and the last instance of \( A \) to generate all other elements. This introduces an additional complexity into the dependence analysis. The dependence is expressed explicitly in this work as

\[ A2 \left[ <-A1 \right] (7) = X. \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]

\[ \nonumber \]
Array methods for straight line and conditional code follow the methods developed for scalars (Figure 9).

1) if A(I) > 100 then
2) A(I+1) = 0
3) else
4) A(I) = 3
5) end if

a) original code

1) if A1(I1) > 100 then
2) A2 [<-A1] (I1+1)=0
   A4 = A2
3) else
4) A3 [<-A1] (I) = 3
   A4 = A3
5) end if

b) single-valued code

Figure 9: Arrays in conditionals

Just as scalars renamed in loops become loop-bounded arrays, so arrays found in loops become arrays of arrays. A new array object is created for each assignment to an array in the loop, and the object number is assigned equal to the iteration variable driving the loop. This iteration id is shown as a superscript in Figure 10†††.

1) do I = 1,N
2) A(I) = A(I) + 3*B(2*I)
3) enddo

a) original code

A2^0 = A1
ALLOC (N1)
do LS = 1,N1
1) I1(LS) = LS
3) enddo
A3 = A2^N1

b) single-valued code

Figure 10: Arrays in loops

††An equivalent FORTRAN-legal syntax, copyall(A2,A4) is inserted in the code.
†††Equivalent FORTRAN-legal syntax for A2^LS(I) is A2(LS,I).
discover whether the iteration of the definition could have preceded the iteration of use. If the definition can precede a use, flow dependence must be assumed (conservatively) and no name is changed. However if the definition must follow a use then the use reference name for the array can be replaced by the loop entry name for the array.

1) $X = X + 3$
2) $Y = X / 5 - Y$
3) do $I = 1, N$
4) $A(I) = B(I) + C(I)$
5) $C(I) = D(I) \times X$
6) if ($I \lt N$) then
7)  $E(I) = C(I) + A(I+1)$
8) endif
9) enddo
10) $X = A(N) + C(N)$

a) original code

1) $X_2 = X_1 + 3$
2) $Y_2 = X_2 / 5 - Y_1$
3) $A_0^2 = A_1$
4) $C_0^2 = C_1$
5) $E_0^2 = E_0$
6) alloc (N1)
7) do $LS = 1, N_1$
3)  $I_1( LS) = LS$
4) $A_2^{LS}(<(<A_2^{LS-1}(LS)$)
5) $C_2^{LS}(<(<C_2^{LS-1}(LS)
6) if $LS \lt N_1$ then
7) $E_1^{LS}(<(E_1^{LS-1}(LS)=C_2^{LS}(LS)+A_2^{LS}(LS+1)$
8) endif
9) enddo
10) $X_3 = A_3(N1) + C_3(N1)$

b) single-valued code

Figure 11: Renaming arrays in loops

In the code of Figure 11, three pairs are analyzed, a pair for each of two C uses and for one A use. DEF subscripts and superscripts carry a prime for clarity.

DEF    USE
1) $S_5 \quad C_2^{LS'}(LS')$  \quad $S_4 \quad C_3^{LS-1}(LS)$
a) align subscripts $\Rightarrow LS' = LS$
b) iteration id solution $LS' \leq LS-1$?

Inequality b) becomes $LS \leq LS-1$ which has an empty solution set. The empty solution set proves that $C_2^{LS}$ is never defined between the loop beginning and the use in $S_4$. Therefore the $C_2^{LS-1}$ name in $S_4$ will be changed to $C_1$.

2) $S_5 \quad C_2^{LS'}(LS')$  \quad $S_7 \quad C_3^{LS}(LS)$
a) align $\Rightarrow LS' = LS$
b) solution $LS' \leq LS$?

Inequality b) becomes $LS \leq LS$ which is everywhere true. Therefore the element access is flow dependent in the loop and the name will not be changed.

3) $S_4 \quad A_2^{LS'}(LS')$  \quad $S_7 \quad A_2^{LS}(LS+1)$
a) align $\Rightarrow LS' = LS+1$
b) solution $LS' \leq LS$?

Inequality b) becomes $LS + 1 \leq LS$ which has an empty solution set. Therefore the $A_2^{LS}(LS+1)$ name is changed to $A_1(LS+1)$.

Generalizing the analysis above, we ask: Is there a solution range to the inequality

iterationidDEF \leq iterationidUSE

when constrained by the alignment of subscripts

scriptDEF = scriptUSE?

The relationship between the iteration id and the subscript is linear and is generated by the normalization routine at the beginning of the loop. There are times when no relationship is apparent between the iteration id and the original subscript (e.g., WHILE loops). In such cases this analysis is abandoned. If the equations cannot be solved or the solution set is not empty then the name of use is not changed. However if the solution set is the empty set, then the use name is changed to the loop entry name.

Having removed unnecessary array object uses, the loop is now scanned for objects defined but not used within the loop. $E_1$ and $A_2$ are both such objects in Figure 11. These definitions are treated as a complete array definition and are allowed to occur on one object. $E_1^{LS}$ will therefore be changed to $E_1$. $A_2^{LS}$ will similarly be changed. The resultant code is shown in Figure 12(a), and can now be successfully parallelized or vectorized.

The liberation of arrays in loops can therefore be accomplished by a two step process, first to rename the arrays, then to resolve dependences where possible. In debugging the code below the AVAIL range of arrays $A$ and $C$ are marked with a condition, "if $LS < I$ produce $A_1$, $C_1$ else report $A_2$, $C_2$". So if a breakpoint is set after $S_5$, a request for $C(3)$ in iteration 2 would produce $C_1(3)$, whereas a request for $A(6)$ in iteration 8 would report $A_2(6)$.
1) \( X_2 = X_1 + 3 \)
2) \( Y_2 = \frac{X_2}{5} - Y_1 \)
   \( C_2^0 = C_1 \)
   alloc (N1)
   do LS = 1,N1
   3) \( I_1(LS) = LS \)
   4) \( A_2(LS) = B_1(LS) + C_1(LS) \)
   5) \( C_2^{LS} = \frac{C_2^{LS-1}}{C_2^{LS-1}}(LS) = D_1(LS) \times X_2 \)
   6) if (LS .lt. N1) then
      7) \( E_1(LS) = C_2^{LS}(LS) + A_1(LS+1) \)
   end if
   8) end do
   9) A3 = A2
   C3 = C2^N1
   E2 = E1
10) \( X_3 = A3(N1) + C3(N1) \)

(a) after DEF/USE analysis

(b) parallelized

Figure 12: Effects of DEF/USE analysis

6. Techniques to Implement Name Reclamation

In the name reclamation stage it is assumed that code which started out as sequential code, has been completely renamed and then transformed by parallelizing software. Now it is the object of the name reclamation stage to discover which of the numerous variable names created in the global renaming stage must be allocated storage and which names can be reclaimed.

A value of a variable must be maintained in storage as long as
1) the value is live in the transformed code or
2) the available range of the value in the sequential code is not yet expired.
A value is \textit{live} as long as it has later uses. A value is \textit{available} as long as its associated variable has not been redefined or the scope of the variable has not terminated. The maintenance range, \( R_{MV} \), for each value indicates how long storage has to be maintained and is computed by

\[ R_{MV} = R_{AV} \cup R_{LV} \]

where \( R_{AV} \) is the available range of the value in the sequential code and \( R_{LV} \) is the live range of the value in the transformed code.

\( R_{AV} \) can be calculated by standard live range analysis for a variable name with extensions to include statements up through the point that the variable is redefined in the sequential execution (rather than its last use). In the computation of \( R_{LV} \), it is assumed that the transformed program has been modified for some form of parallel execution. A task with a live range for a value may be assigned to 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.

Two variables are said to have \textit{overlapping} maintenance ranges if they must both be maintained at the same time. In sequential code this means at the same statement number. In parallel code a maintenance range includes a range of statements up through a synchronization point.

The name reclamation algorithm scans the transformed program, looking for pairs of variables with the same root name and disjoint maintenance ranges. This is determined efficiently in practice for a pair of variables, say \( X_1 \) and \( X_2 \) a later definition of \( X \), by testing
1) is \( X_1 \) still live in this task or any concurrent segment or successor?
2) is \( X_1 \) in the \textit{AVAIL} list of any later statement in this task, or any concurrent task or successor?
3) is \( X_2 \) in the \textit{AVAIL} list of any concurrent task statement?
If the answer to all of these questions is NO then \( X_1 \) subsumes \( X_2 \), that is, the \( X_2 \) name disappears and is replaced everywhere by \( X_1 \) (except in synchronization flags). Note that in reclaiming the \( X_2 \) name, it is actually the \( X_1 \) value that is being overwritten.

A high level version of the name reclamation algorithm is shown in Figure 13. It begins by scanning the program in statement order to compute four items:

1) a graph of concurrent ranges where concurrent ranges begin with \textit{PARBEGIN}, \textit{DOALL} or other structured parallel constructs,
Algorithm RECLAIM_NAMES (P: procedure)

Load Database of avail sets
Produce Program Dependency Graph
{ nodes are blocks of code and concurrency is indicated}
Compute "last use" and "last availability" of each variable
Compute set of AVAIL names for each block,
   AVAIL = union of AVAIL sets for each statement
   associated with the block

Replace_Name_List = empty
ACTIVE_SET = empty
{ACTIVE_SET = variables defined, reaching block}

For each BLOCK do (in statement order)
   Check AVAIL set against Replace_name_list
   If any are found then
      replace with Replace_name
   end for USE
   For each statement in BLOCK do
      For each USE
         Check each Use against Replace_name_list,
         if found update with Replace_name
      end for USE
      For each DEF
         Repeat for ACTIVE_SET with matching rootname
         if (ACTIVE_SET ext# not live in any
            parallel segment or successor)
            {sufficient to check if last use < present position
             and last use not in parallel segment *}
            and ( ACTIVE_SET # not in AVAIL set
            of any parallel segment or successor)
            {sufficient to check if last avail < present position
             and last avail not in parallel segment *}
            and (Def ext # not in AVAIL of any paral. segment)
            then
               change DEF ext # to ACTIVE_SET ext #
               add entry to Replace_Name_list
               (e.g. Def ext #$=ACTIVE_SET ext #)
         endif
      end for DEF
   Until DEF is reclaimed or ACTIVE_SET is exhausted
   {Loop entry and exit names are reclaimed when
   associated looping variables are reclaimed}
   *[Locally computed last use and last avail sets are
   used between IF and ELSE]
   If not reclaimed add DEF ext # to ACTIVE_SET
end for BLOCK
Update Procedure parameter lists
Write updated Database
end RECLAIM_NAMES

Figure 13: Name Reclamation Algorithm

2) the last lexical statement number of use of each variable in the program,
3) the last lexical statement where each variable is available in the program (this is computed by comparing the present program order with the AVAIL sets stored on the Database),
4) a block AVAIL set for each block that is the union of all variables available within that block.

The algorithm then rescans the program to reclaim names. Names that are not reclaimed become part of the ACTIVE_SET. Names that are reclaimed will produce a name replacement directive in the Replace_Name_List. Figure 14 shows the effect of applying name reclamation to the parallelized program seen in Figure 12.

1) X1 = X1 + 3
2) Y1 = X1 / 5 - Y1
   C2 = C1
   alloc (N1)
   forall LS in 1..N1 do
3) I1(LS) = LS
4) A2(LS) = B1(LS) + C1(LS)
5) C2(LS) = D1(LS) * X1
6) if (LS < N1) then
7)   E1(LS) = C2(LS) + A1(LS+1)
8) end if
9) end do
10) X3 = A2(N1) + C2(N1)

Figure 14: The results of Name Reclamation

The number of names reclaimed depends largely on the useful parallelism exploited in the program. Highly parallel programs reclaim few names. On the other hand, in the case of a sequential program which was liberated but not transformed for parallelism, all of the added names are reclaimed.

Revisiting the example in Figure 1, it is now possible to answer the questions posed in the introduction.

T1=0
(1) T2(0) = T1
(2) I1(1..M1) = 1..M1
   do 99 LS1 = 1, M1
(3)   T2(LS1) = T2(LS1)+sqrt(LS1)
   99 continue
   do 100 LS2 = 1, N1
(4)   J1 = LS2
(5)   A1(1..M1,LS2)=A1(1..M1,LS2-1)/T2(LS1)
   100 continue

Figure 15: Debugging the example of section 1
Figure 15 shows the example after global renaming, parallelization and name reclamation have been applied. The questions "What is T?" or "What is I?" at a breakpoint inserted after statement 4 would cause T2(LS 1) or I1(LS 1) to be examined, where LS1 is the iteration number of the outer loop. The user will be queried as to which iteration number is desired before reporting the values.

7. The Runtime Interface

Each variable not reclaimed represents a storage location whose values are current in the parallel program whenever the associated values are current in the original sequential program. However it is possible to request a value at runtime before the value is ready. This can be true in the case of parallel tasks (as in the request for Z in Figure 3) and also in the case where code is moved forward. In such cases it is necessary for the runtime interface to distinguish between defined and undefined variables.

One solution which could be implemented on any architecture is to start each program segment, subroutine or main, with storage initialized to binary zero. The debugger can check for this value, and warn the user that the value requested is either not yet available or is equal to zero. A more sophisticated approach is available if synchronization bits guard each word on the hardware. Some architectures also enforce a write-before-read in the hardware. This cell-by-cell low level synchronization is available on the HEPIO and in the I-structure memory in dataflow2.

8. Implementation and Experimentation

The prototype system presently under development as the basis for this research is written in C for the Sun Workstation platform. The code liberation package is about 2000 lines while reclamation is about 700 lines of code. It is capable of liberating structured FORTRAN 77 scalar code for all statement types. FORTRAN code containing COMMON variables is excluded in this prototype. In the current system, arrays are passed through without being renamed. Additional development is currently underway to implement the automatic renaming of array types. Hand renaming of arrays has been done to prove the feasibility of array renaming and to provide the measures of storage expansion shown in Figure 16. FORTRAN programs chosen for experimentation range from small (25 lines) to medium large (about 400 lines). They were renamed, hand parallelized or vectorized and names were reclaimed to obtain the data below. Eight processors are assumed. In the figure the size of the storage requirement is reported in words. Loop limits are assumed to be the maximum possible.

<table>
<thead>
<tr>
<th>FORTRAN programs</th>
<th>Primer</th>
<th>Longmult</th>
<th>Str3</th>
<th>Len301</th>
</tr>
</thead>
<tbody>
<tr>
<td>length (lines)</td>
<td>25</td>
<td>52</td>
<td>253</td>
<td>402</td>
</tr>
<tr>
<td>original storage</td>
<td>3</td>
<td>85</td>
<td>810</td>
<td>1.8K</td>
</tr>
<tr>
<td>after renaming (% incr)</td>
<td>45</td>
<td>1838</td>
<td>26K</td>
<td>58K</td>
</tr>
<tr>
<td></td>
<td>(1500%)</td>
<td>(2000%)</td>
<td>(3300%)</td>
<td>(3200%)</td>
</tr>
<tr>
<td>after reclmtn (% incr)</td>
<td>26</td>
<td>180</td>
<td>2.1K</td>
<td>9.8K</td>
</tr>
<tr>
<td></td>
<td>(866%)</td>
<td>(112%)</td>
<td>(260%)</td>
<td>(550%)</td>
</tr>
<tr>
<td>incr due to parallelztn</td>
<td>99%</td>
<td>80%</td>
<td>89%</td>
<td>94%</td>
</tr>
<tr>
<td>incr due to debugging</td>
<td>1%</td>
<td>20%</td>
<td>11%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Figure 16: System imposed memory expansion

These preliminary statistics show estimated increases in storage size to be on the order of 3 to 5 times the original storage. Close analysis reveals that about 90% of this increase is due to names not reclaimed because they enabled useful parallelism (e.g., local copies of variables). The other 10% are names not reclaimed so that non-current values could be correctly reported by the debugger. Since the storage enlargement is so dominated by the exploitation of useful parallelism, it is probable that the enlargement is bounded by the number of available processors or the extent of parallelism in the application, whichever is smaller.

The third stage of the system, the runtime interface, will be implemented last. It will take the form of a small subroutine added to an existing debugger which can trap and replace variable requests via access to the database.

As the prototype system reaches completion, it will be possible to collect data on a wide variety of FORTRAN code, and interact with a variety of parallelizing software. The present implementation of name reclamation handles the structured parallel constructs for both MIMD and SIMD architectures including vector operations, DOALL and ParBegin/ParEnd.
9. Conclusions

This work envisions a more modular parallel program development environment, one which is characterized by less architectural dependence than is now commonly the case. In such an environment portable sequential code would be parallelized by alternate software packages for different architectures. If values made non-current by such transformations are to be reportable in debugging sessions, the techniques employed should be independent of the choice of transformational package (and target architecture).

This work enables reasonable debugging of such transformed code to proceed without fear of misreporting noncurrent variable values. This is a problem indigenous to the debugging of code transformed for any reason, whether for optimization or for parallelization. Therefore this work provides a more general solution to the problem of debugging optimized code which has been studied earlier.

The merit of debugging the final form of the program (as opposed to the sequential source code) has been well-established in the case of optimized code. Such arguments are even stronger when code has been parallelized. The number and complexities of such transformations added to the involvement of additional processors and process communication all increase the risk that a bug may be introduced. In addition, the length of the program running sequentially may make sequential debugging impractical. Yet another consideration is introduced when a real-time response is required. In such a system, debugging the final parallel form may be mandated.

In the debugging session of the transformed code, this work will allow values to be reported that the programmer would have expected to be current at the breakpoint position in the sequential code. In cases where the appropriate value is unavailable, the method will report unavailability as opposed to an incorrect or misleading value.

This result is achieved through the creation of single assignment form for the input code, a form which has several additional advantages as well as liabilities. The primary liability, the increase of storage consumed, has been resolved by this work by reclaiming the unneeded names after the transformations are completed. This allows the advantages of the single-assignment form to be freely used.

References

1. J.R. Allen and Ken Kennedy, "PFC: A Program to Convert Fortran to Parallel Form", Rice University MASC TR82-6, March 1982.


