ANL-85-63 168 6-24-86 I-26728 JT (1) 40

10222

ANL-85-63
DR-1778-7

TRANSFORMING FORTRAN DO LOOPS TO IMPROVE PERFORMANCE ON VECTOR ARCHITECTURES

by

Wayne R. Cowell and Christopher P. Thompson

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Operated by THE UNIVERSITY OF CHICAGO
for the U. S. DEPARTMENT OF ENERGY
under Contract W-31-109-Eng-38

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Distribution Category:
Mathematics and Computers
(UC-32)

ANL-85-63

MASTER

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

ANL--85-63

DE86 012086

Transforming Fortran DO Loops to Improve Performance on Vector Architectures

Wayne R. Cowell

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois

and

Christopher P. Thompson

Computer Science and Systems Division
Atomic Energy Research Establishment
Harwell, Oxfordshire
on leave at Numerical Algorithms Group, Inc., Downers Grove, Illinois

May 1986

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Contents

Abstract

The performance of programs executing on vector computers is significantly improved when the number of accesses to memory can be reduced. Unrolling Fortran DO loops, followed by substitutions and eliminations in the unrolled code, can reduce the number of loads and stores. In this paper we characterize the unrolling transformation and associated transformations of Fortran DO loops and describe a set of software tools to carry out these transformations. The tools use the machinery available in Toolpack and have been integrated into that environment. We describe the results of applying these tools to a collection of linear algebra subroutines.

Transforming Fortran DO Loops to Improve Performance on Vector Architectures*

Wayne R. Cowell†
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois

Christopher P. Thompson
Computer Science and Systems Division
Atomic Energy Research Establishment
Harwell, Oxfordshire
on leave at
Numerical Algorithms Group, Inc.
Downers Grove, Illinois

1. Introduction

Fortran[1] programs that include matrix and vector calculations usually call highly optimized linear algebra routines from a library; see, for example, [7,4]. The nested DO loops in such linear algebra routines have properties that make the code amenable to the application of certain transformations, described in [5], aimed at causing a vectorizing Fortran compiler to produce code that is better adapted to vector architecture. The aim of the paper is to describe preliminary work in the automation of these transformations. We begin by illustrating the transformations and their effect.

Consider the computation of a matrix by vector product ($c = Ab$):

```
      DO 20 J=1,N
         DO 10 I=1,64
               C(I) = C(I) + A(I,J)*B(J)
10       CONTINUE
20    CONTINUE
```

The following generic "pseudo-assembler" illustrates the code generated from the assignment statement in the above Fortran by a typical vectorizing compiler for a register-based machine:

---

*This report is being issued jointly as Argonne National Laboratory Technical Report ANL-85-63 and Numerical Algorithms Group Technical Report NP 1168.

†This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

[1]The word *Fortran* in this paper means ANSI Standard Fortran 77 [1].

```
1       load c(1:64)    into vector register V0
2       load a(1:64,j)  into vector register V1
3       load b(j)       into scalar register S0
4       * S0,V1         into vector register V1
5       + V1,V0         into vector register V0
6       store V0        into c(1:64)
```

For simplicity we shall ignore the looping machinery generated by the compiler—it is computationally negligible by comparison with the arithmetic shown.

The following Fortran code is exactly equivalent, from an arithmetic viewpoint, to the above code. (For clarity we assume that N is even.) The differences between the loops only affect data movement; all the calculations, including intermediate values, are identical.

```
      DO 20 J = 1,N-1,2
          DO 10 I = 1,64
              C(I) = (C(I) + A(I,J)*B(J)) + A(I,J+1)*B(J+1)
10        CONTINUE
20    CONTINUE
```

From the second version of the assignment statement, our hypothetical vectorizing compiler generates

```
1       load c(1:64)      into vector register V0
2       load a(1:64,j)    into vector register V1
3       load b(j)         into scalar register S0
4       * S0,V1           into vector register V1
5       + V1,V0           into vector register V0
6       load a(1:64,j+1)  into vector register V2
7       load b(j+1)       into scalar register S1
8       * S1,V2           into vector register V2
9       + V2,V0           into vector register V0
10      store V0          into c(1:64)
```

The significant fact is that this second set of assembler instructions is executed only N/2 times while the first set is executed N times; hence, the total number of vector load and store instructions is reduced. With present computer technology this fact is extremely important because memory access is the bottleneck for most computations.

In principle we can further reduce the number of vector loads and stores by *unrolling* to greater *depth* (i.e., changing DO 20 J=1,N-1,2 to DO 20 J=1,N-d+1,d where d > 2) and suitably modifying the assignment statement. (For convenience, we assume that d divides N but, as will be seen later, our treatment does not rely on this.) In the set of instructions generated by the compiler the arithmetic operations are the same but the set is executed N/d times. The number of words moved between memory and registers in vector and scalar operations is

Words Moved

Vector stores:	64N/d
Vector loads:	(64N/d)(d+1)
Scalar loads:	64N

The most expensive operations are vector loads and stores. The number of words moved by vector instructions, for various values of d, is

d	Words Moved = 64N(d+2/d)
1 (original)	192N
2	128N
4	96N
64	66N

Because it is not meaningful for d to be larger than N, the lower limit is 64(N+2)—about one-third the number for the original code. The cost of other types of instructions and the finiteness of the hardware make this limit unattainable, but we shall present the results of experiments which show that a significant proportion of this improvement can be achieved. It is also interesting to note that the largest single improvement is obtained [2] when d = 2.

Other benefits from such transformations of Fortran are worth noting. These are harder to quantify and depend, to some extent, on the sophistication of the compiler; but it seems reasonable to expect computer manufacturers to supply sufficiently sophisticated software to utilize the features that have been built into the hardware. There are indications that some compiler writers are beginning to consider this type of transformation.

First we note that reducing the number of memory references decreases the probability of delays introduced by memory latency time. Further, in a multiprocessor environment, a reduction in the number of memory references decreases the probability of conflicts among processors, thereby lessening the delays caused by such conflicts and increasing the speed of all executing processes (a socially desirable benefit!).

Benefits also derive from the fact that many existing vector computers have a limited amount of parallelism incorporated into their architecture and the transformed code provides enhanced opportunities for concurrency. For example, the second set of load instructions (6 and 7) is independent of and could be scheduled concurrently with the first set of load instructions (2 and 3), provided there were enough paths to memory, or with the preceding arithmetic instructions (4 and 5). The possibilities for simultaneous execution of arithmetic operations are also enhanced. The only restriction is that there be enough vector registers to contain the data. If this is not the case, then presumably either the compiler will issue an error message (this has been observed when d is large) or will generate code roughly like the first set.

We have written software tools that transform the parameters and ranges of a class of Fortran DO loops in the manner of the example. Given such tools in the programming environment for a vector computer, library routines may be written and maintained in their original, more compact, form while the task of generating transformed versions is given to the

[2] Although this is true in our simple model, the actual decrease in execution time depends on the architecture of the machine. For example, if there are two paths from memory, and they can function concurrently, the largest gains are observed when unrolling to an odd depth. The reason can be detected by observing that instruction 6 (load a(1:64,j+1)) in the second set leaves one path from memory idle but could be paired with load a(1:64,j+2) if the unrolling depth were 3.

tools. Programmers can easily generate transformed versions, unrolled to various depths, enabling them to perform experiments that determine the optimal unrolling depth for a particular routine on a particular machine configuration. Additionally, the tools may serve as models for compiler writers who wish to incorporate such transformations into compilers for vector machines. This is part of a longer-term effort to construct tools that transform existing programs so that their performance is enhanced on novel computer architectures.

We have concentrated initially upon full linear algebra subroutines for matrix and vector operations. Human experts organize these calculations in terms of matrix by vector operations. The transformed code can then be viewed as the implementation of a new algorithm (whose validity may be demonstrated mathematically) that performs several vector operations on each pass through the outer loop. On the other hand, no knowledge about computational linear algebra is explicitly built into the tools. They are capable of detecting patterns and manipulating programs using algorithms that reflect knowledge of the target architecture. Therefore, the analysis underlying the tool algorithms concentrates on operations that may be performed on Fortran programs to improve their performance on a particular architecture. To produce the transformed Fortran discussed in this paper, several intermediate stages of a program (created by different transformations) must be introduced before the "final version" that exhibits the improved data movement. These are described in Sections 2-5.

In Section 2 we define formal restrictions on the domain of the transformations. Although Fortran DO loops can be extremely complex, many linear algebra operations can be coded using a subset of the language - often such restrictions also improve efficiency. The concept of a *regular* DO loop, discussed in Section 2, admits sufficient generality to allow coding a full range of linear algebra routines (and many other routines) in a format that is amenable to various transformations.

In the first intermediate stage of transformation, the innermost-but-one DO loop of each nest is *unrolled;* i.e., the range of the loop, including, of course, the inner DO loops, is reproduced d times with appropriate substitutions for the DO variable and appropriate modification of the parameters. (Our assumption is that the arithmetic is efficiently handled by vector instructions.) At this stage, related assignment statements, in the ranges of the inner DO loops, are generated for further processing. We have concentrated on routines with DOs nested two deep, and so the loop that is unrolled by the tools is actually the outer loop.

In the example of matrix by vector multiplication the code generated at this first stage when d = 2 (assuming that N is even) is

```
      DO 30 J = 1,N-1,2
           DO 10 I = 1,64
                C(I) = C(I) + A(I,J)*B(J)
10         CONTINUE
           DO 20 I = 1,64
                C(I) = C(I) + A(I,J+1)*B(J+1)
20         CONTINUE
30    CONTINUE
```

The unrolling transformation is discussed in Section 3.

In the second stage of intermediate processing the two DO loops above are *condensed* to give

```
      DO 30 J = 1,N-1,2
          DO 10 I = 1,64
              C(I) = C(I) + A(I,J)*B(J)
              C(I) = C(I) + A(I,J+1)*B(J+1)
10        CONTINUE
30    CONTINUE
```

We have aggregated the assignments that were generated from the same "master."

In the third stage the two assignments are combined by *substitution/elimination* to give the transformed version from which the more efficient assembler code was generated, as analyzed above. Conditions prerequisite to condensation and substitution/elimination are discussed in Section 4.

In Section 5 we work through the transformations for a typical program paradigm. Then, in Section 6, we describe the tools that perform these transformations and how they are integrated into Toolpack [2]. In Section 7 we report the results of timing experiments in which we compare machine-transformed with human-transformed versions of the codes described in [4]. Finally, in Section 8, we suggest future directions for this work.

2. Regular DO Loops

In the following, I is a variable of type INTEGER, F(I) is a block of Fortran statements that may depend on I, t(I) is a single Fortran statement that may depend on I, and e1,e2,e3 are expressions of type INTEGER. The Fortran DO loop

```
        DO 10 I = e1,e2,e3
           F(I)
10         t(I)
```

is said to be *regular* when the following conditions hold:

(1) The terminating statement t(I) is a CONTINUE.

(2) There are no transfers to the terminating statement from the block F(I).

(3) If the parameters are such that the loop is executed at least once, then

$$e2 = e1 + m*e3$$

for some integer m.

(4) If the loop terminates by "dropping through" the terminating statement (rather than by transferring out of the loop) then the value of the DO variable is not used after the termination of the loop.

The following lemma will be useful. Its proof relates condition (3) to loop processing as defined in [1].

LEMMA 1. *Regularity condition (3) is equivalent to saying that the execution of*

```
        DO 10 I = e1,e2,e3
           F(I)
10      CONTINUE
```

is equivalent to the execution of the sequence

```
F(e1)
F(e1+e3)
F(e1+2*e3)
   .
   .
   .
F(e2)
```

Proof. The Fortran standard [1] defines the initial iteration count to be

$$IR = MAX((e2 - e1 + e3)/e3,0)$$

where "/" is integer divide in the Fortran sense. The iteration count is decremented by 1 each time the loop is executed; execution of the loop continues as long as the iteration count is non-zero. If IR > 0 (that is, if the loop is executed at least once), there is an integer r such that $0 \le r < e3$ and

$$e2 - e1 + e3 = IR*e3 + r$$

If we assume regularity condition (3), then

$$e2 - e1 + e3 = (m + 1)*e3$$

and so r = 0 and IR = m + 1. Corresponding values of the iteration ccunt and the DO variable are as follows:

IR	e1
IR - 1	e1 + e3
IR - 2	e1 + 2*e3
...	...
IR - (IR -1) = 1	e1 + (IR - 1)*e3 = e2

The sequence of values assumed by the DO variable corresponds to the execution of F(e1), F(e1+e3), ..., F(e2), as required.

Conversely, if the execution of F(e1), F(e1+e3), ..., F(e2) is equivalent to the execution of the DO, then

$$e2 = e1 + (IR - 1)*e3$$

and regularity condition (3) is satisfied. *Q.E.D.*

A DO loop that violates one or more of conditions (1)-(3) may be transformed into equivalent Fortran in which the DO loop satisfies these conditions. Let loop 10, violating conditions (1)-(3), be represented by

```
      DO 10 I = e1,e2,e3
         F(I) [10]
10       t(I)
```

where "[10]" indicates transfers to the statement with label 10 from the block F(I). Then the following code is equivalent to loop 10; further, loop 11 satisfies conditions (1)-(3):

```
      IR = (e2 - e1 + e3)/e3
      E2NEW = e1 + (IR - 1)*e3
      DO 11 I = e1,E2NEW,e3
         F(I) [9]
 9       CONTINUE
         t(I)
 11    CONTINUE
```

(We are assuming that IR and E2NEW are not used elsewhere in the program.) To verify equivalence, note that if loop 11 is executed at least once, then IR is the initial iteration count and the execution of loop 11 is equivalent to the execution of

```
         F(e1) [1]
 1       CONTINUE
         t(e1)
         F(e1+e3) [2]
 2       CONTINUE
         t(e1+e3)
         F(e1+2*e3) [3]
 3       CONTINUE
         t(e1+2*e3)
            .
            .
            .
         F(E2NEW) [<label>]
<label>  CONTINUE
         t(E2NEW)
```

By construction, E2NEW is the same last value as before. Note also that condition (3) is true by Lemma 1.

Condition (4) may seem unduly restrictive in view of the fact that it is easy to adjust the "drop-through" exit value of I by following

```
 11       CONTINUE
```

in the above code with

```
      IF (IR .LE. 0) THEN
         I = e1
      ELSE
         I = E2NEW + e3
      END IF
```

However, this appended code inhibits the application of certain transformations to be discussed later. Moreover, our experience with codes that perform matrix and vector operations suggests that the final value of the DO variable is rarely used.

Thus it is clear that any DO loop can be automatically transformed into a regular DO loop. The regular DO loops form the domain and range of the unrolling and condensing transformations defined in Sections 3 and 4.2, respectively. Observe that if the loop contains labelled statements, then extra care is required to avoid clashes when generating new code. The necessary modifications are straightforward, and we do not discuss them explicitly in the following.

3. The Unrolling Transformation

The transformation considered in this section is a generalization of the unrolling transformation discussed in [6] and pursued further in [5]. The aim here is to generate sequences of assignment statements that can be combined by subsequent transformations. As illustrated in Section 1, a DO loop is unrolled by replicating the statements in its range and suitably adjusting the parameters. In general there will be values of the DO variable not assumed by the new set of parameters and it will be necessary to have a "clean-up" loop to cover these cases. We can choose the new parameters so that the cases covered by clean-up occur either at the end or at the beginning of the set of values assumed by the DO variable. We call these alternative constructions "clean-up after" and "clean-up before," respectively. The tools use clean-up after and we concentrate on that case in the following analysis.

The original (or "rolled") loop, the unrolled loop, and the clean-up loop (called loops 10, 20, and 21, respectively) have the following forms, where d, the depth of the unrolling, is a positive integer.

```
C This is the original loop.
      DO 10 I = e1,e2,e3
         F(I)
10    CONTINUE

C This is the main unrolled loop.
      DO 20 I = e1main,e2main,d*e3
         F(I)
         F(I+e3)
            .
            .
            .
         F(I+(d-1)*e3)
20    CONTINUE

C This is the clean-up loop.

      DO 21 I = e1clup,e2clup,e3
         F(I)
21    CONTINUE
```

Define

$$M = (e2 - e1 + e3)/(d*e3)$$

where e1,e2,e3 are the parameters of loop 10. Then the parameters of loops 20 and 21 are

defined as follows:

e1main = e1, e2main = e1 + (M - 1)*d*e3
e1clup = e1 + M*d*e3, e2clup = e2

THEOREM 1. *If loop 10 is regular, then loops 20 and 21 are regular and together are equivalent to loop 10.*

Proof. Regularity condition (1) is true by construction for loops 20 and 21. We shall first establish regularity condition (3) for these loops and then show that together they are equivalent to loop 10. Regularity conditions (2) and (4) will follow from the equivalence.

For loop 20 we note that

$$e2main = e1main + (M - 1)*d*e3$$

and thus regularity condition (3) is satisfied. For loop 21,

$$e2clup = e1clup + e2 - e1 - M*d*e3.$$

Loop 10 is regular, and so e2 - e1 = m*e3 for some integer m. Hence,

$$e2clup = e1clup + (m - M*d)*e3$$

and regularity condition (3) for loop 21 is satisfied.

Next we establish that loops 20 and 21 are equivalent to loop 10. By Lemma 1 the main loop is equivalent to the blocks in the first column below. Corresponding values of the main loop variable are shown in the second column (only changes are recorded).

```
F(e1)                       I = e1main = e1
F(e1+e3)
   .
   .
   .
F(e1+(d-1)*e3)
F(e1+d*e3)                  I = e1 + d*e3
F(e1+(d+1)*e3)
   .
   .
   .
F(e1+(2*d-1)*e3)
   .                           .
   .                           .
   .                           .
F(e2main)                   I = e2main
F(e2main+e3)
   .
   .
   .
F(e2main+(d-1)*e3)
```

By Lemma 1 the clean-up loop may be similarly depicted:

```
F(e1clup)                 I = e1clup
F(e1clup+e3)              I = e1clup+e3
   .                              .
   .                              .
   .                              .
F(e2)                     I = e2clup = e2
```

Consider the last block in the main loop and the first block in the clean-up loop. From the new parameter definitions

$$e2main + (d-1)*e3 = e1 + M*d*e3 - e3 = e1clup - e3$$

Hence, the concatenation of the first columns is equivalent to loop 10. Regularity conditions (2) and (4) for loops 20 and 21 follow immediately. *Q.E.D.*

We note that the clean-up loop is not executed when e2 - e1 + e3 is exactly divisible by d*e3. To show this, let the initial iteration count for the clean-up loop be IRclup. Then,

$$IRclup = MAX((e2clup - e1clup + e3)/e3,0) = MAX((e2 - e1 + e3 - M*d*e3)/e3,0)$$

If e2 - e1 + e3 is exactly divisible by d*e3, then from the definition of M, M*d*e3 = e2 - e1 + e3 and IRclup = 0.

For clean-up before, loop 21 would precede loop 20, M would be defined as before, and the new parameters would be

$$e1main = e2 - (M*d - 1)*e3,\ e2main = e2 - (d-1)*e3$$

$$e1clup = e1,\ e2clup = e2 - M*d*e3.$$

The above analysis can be carried through for clean-up before without essential change.

4. Dependency Sets

Any regular DO loop may be unrolled, as seen in Section 3, but in order to condense sequences of DO loops and then combine the statements in the range of the condensed DO loop, we need the transformations discussed in this section. The independence conditions that permit these transformations to be applied without changing the results of the computation are expressed using the notion of the *dependency set* of an assignment statement. We shall limit the discussion to arithmetic assignments because of our focus on modules that carry out vector and matrix computations. To simplify the discussion and the operation of the tools, we shall exclude statements that contain function references and shall require that the program contain no EQUIVALENCE statements.

Let $v = e$ be an arithmetic assignment statement. Then v is the name of a variable or an array element of type integer, real, double precision, or complex, and e is an arithmetic expression that, by assumption, does not contain a function reference [1]. The dependency set of $v = e$ is the set of memory locations associated with the variable and array references in v and e.

In other words, if we regard $v = e$ as a definition of v, then this definition is dependent on the definitions of exactly the members of the dependency set, including, for convenience, the location associated with v itself. To illustrate, the dependency set of each arithmetic assignment statement in the first column below is the set of locations associated with the references in the second column:

```
A = 2                          [A]
A = B + C(I,J)                 [A,B,I,J,C(I,J)]
S = S + 1                      [S]
C(A(I,J)) = 2                  [I,J,A(I,J),C(A(I,J))]
C(I) = A(I,J)*B(J)             [I,C(I),J,A(I,J),B(J)]
C(I) = C(I)+A(I,J)*B(J)        [I,C(I),J,A(I,J),B(J)]
C(2) = C(2)+A(2,J)*B(J)        [C(2),J,A(2,J),B(J)]
```

We shall need to be able to decide whether a variable or array element is a reference to a member of the dependency set of an assignment statement. Even by excluding Fortran EQUIVALENCE, the question is not necessarily settled by examining the names in Fortran expressions because two array references with the same name, but differently named subscripts, refer to the same memory location if the subscripts are equal. The tools examine names and could determine, for example, that the following pairs of array references are different:

C(I) and C(I+1)
C(2,I) and C(1,J)
B(I) and C(J)

However, without further information, the tools could not determine whether or not C(I) and C(J) were different references. Rather, the tools detect whether the assignment statements and the array references in question arise in the context of a Fortran structure in which the subscripts have been analytically shown to be different. Such a structure is analyzed in Section 5.

4.1. Permutation and Substitution/Elimination

Using the notion of dependency sets, we can state the conditions under which the transformations *permutation* and *substitution/elimination* may be performed (without changing the final result). In the following, an *Assignment Block* is a sequence of arithmetic assignment statements containing no function references and no assignments to subscripts of array references in the block. By *fixed subscripts* we refer to program execution; the value of the subscript has been assigned before the execution of the assignment block and, since there are no assignments to array subscripts in the block, does not change value during the execution of the block. Thus, the fixed subscripts behave mathematically as constants, but we shall reserve the term "constant" to refer to a Fortran constant. The set of allowable values depends on the Fortran context of the assignment block; in our applications, the allowable values are the values assumed by DO variables.

Permutation: The assignment block consisting of

$$u_1 = v_1$$
$$u_2 = v_2$$

may be replaced by

$$u_2 = v_2$$
$$u_1 = v_1$$

if, for every allowable set of fixed subscripts, (1) u_1 is not a reference to a member of the dependency set of $u_2 = v_2$; (2) u_2 is not a reference to a member of the dependency set of $u_1 = v_1$.

Substitution/elimination: Let $u = redef$ be a redefinition of $u = def$ in the assignment block

$$u = def$$
$$u_1 = v_1$$
$$\vdots$$
$$u_i = v_i$$
$$\vdots$$
$$u_n = v_n$$
$$u = redef$$

Suppose that, for every allowable set of fixed subscripts, none of the u_i is a reference to a member of the dependency set of $u = def$. Then the above sequence may be replaced by

$$u_1 = v_1\ [u \rightarrow def]$$
$$\vdots$$
$$u_i = v_i\ [u \rightarrow def]$$
$$\vdots$$
$$u = redef\ [u \rightarrow def]$$

where the first definition, $u = def$, has been eliminated and every occurrence of u on the right-hand side of any assignment statement in the sequence has been replaced by the expression *def*, suitably parenthesized. Note that the number of arithmetic operations in the transformed code is never smaller than in the original, but is the same when precisely one substitution occurs. One substitution is typical for linear algebra codes.

4.2. Condensation of Regular Parameter-Equivalent DO Loops

Condensation is a technique for coalescing the ranges of DO loops, thus creating sequences of assignments with the potential for substitution/elimination. The most basic case of condensation is considered in this section. The tools attempt to reduce more complex cases to this one, as we shall illustrate in Section 5.

Let

```
      DO 10 I = e1,e2,e3
        F(I)
10    CONTINUE
      DO 20 I = e1,e2,e3
        G(I)

20    CONTINUE
```

be a pair of consecutive regular DO loops that have the same parameters (we use the term *parameter-equivalent)*, where F(I) and G(I) are assignment blocks. We wish to condense the two DO loops into one whose parameters are the same and whose range is the concatenation of the two ranges. The condensed loop is

```
      DO 30 I = e1,e2,e3
          F(I)
          G(I)
30    CONTINUE
```

The following theorem provides a sufficient condition for this condensation to take place.

THEOREM 2. *Let k, c be any allowable values of the DO variable such that k*e3 < c*e3. Let a = f [I → c] and b = g [I → k] be any statements from the first and second ranges, respectively, in which the values c and k have been substituted for the DO variable I. Then loop 30 is equivalent to loops 10 and 20 together if a is not a reference to a member of the dependency set of b = g [I → k] and b is not a reference to a member of the dependency set of a = f [I → c].*

Proof. By regularity, loops 10 and 20 together are equivalent to the following assignment block in which all the subscripts are fixed:

```
F(e1)
F(e1+e3)
    .
    .
    .
F(e2)
G(e1)
G(e1+e3)
    .
    .
    .
G(e2)
```

We proceed by induction on n where e1 + (n-1)*e3 is an allowable value of the DO variable. For the case n = 1, write the above sequence of blocks as

```
F(e1)
   .
   .
   .
F(e1+r*e3)      r > 0
   .
   .
   .
F(e2)
G(e1)
   .
   .
   .
G(e2)
```

Choose k = e1 and c = e1 + r*e3. Then

$$k*e3 = e1*e3 < e1*e3 + r*(e3)**2 = c*e3$$

and the hypothesis of the theorem guarantees that block G(e1) can be permuted, statement by statement, with each higher (i.e., previous) block until the next higher block is F(e1).

Now suppose that the first n blocks associated with loop 20 have been permuted upward so that the sequence of blocks may be written

```
F(e1)
G(e1)
   .
   .
   .
F(e1+(n-1)*e3)
G(e1+(n-1)*e3)
F(e1+n*e3)
   .
   .
   .
F(e2)
G(e1+n*e3)
   .
   .
   .
G(e2)
```

Choose k = e1 + n*e3 and c = e1 + m*e3 where m > n > 0. Then

$$k*e3 = e1*e3 + n*(e3)**2 < e1*e3 + m*(e3)**2 = c*e3$$

and the hypotheses of the theorem guarantees that block G(e1+n*e3) may be permuted, statement by statement, with each higher (i.e., previous) block until the next higher block is F(e1+n*e3). By induction, we may permute all the blocks associated with loop 20, giving

```
F(e1)
G(e1)
  .
  .
  .
F(e2)
G(e2)
```

which is equivalent to loop 30. *Q.E.D.*

5. A Typical Paradigm

We now illustrate the use of the transformational machinery described in the previous two sections. We shall step through the stages described in the Introduction for a nest of DO loops that matches one of the paradigms that the tools recognize and transform.

The seven special paradigms that the tools currently recognize were derived from the Fortran structures that occur in several generic linear algebra routines, for example, the routines of [4]. The tools also transform more general structures using general algorithms; they issue warnings when this occurs because data dependencies may render the code incorrect and, in any case, its performance is apt to be poor. When this happens, it is a signal that an additional paradigm needs to be incorporated. Further information may be found in [3].

In the following, an *Array Assignment Block* (AAB) is an assignment block in which the left side of each statement is an array reference.

The paradigm is the nest

```
      DO 10 J = 1,N
         DO 20 I = J+1,M
            B(I) = F(I,J)
20       CONTINUE
10    CONTINUE
```

where B(I) = F(I,J) is an AAB in which the left side of each statement has a single subscript which is the DO variable. The right side of each statement depends on the DO variables I and J; any other subscripts are considered fixed. We require the AAB to have the *restricted subscript property*, defined as follows: Let b(I) = f(I,J) be any statement in the block. Then any occurrence of the array reference b on the right side of any statement in the block has subscript J, except that b(I) is also permitted on the right side of a statement whose left side is b(I).

Completing the description of the paradigm, we require that the left sides of the statements in the block be distinct. However, there is no loss of generality in this requirement, for suppose there were two occurrences of b(I) on the left, either consecutive or separated by statements whose left sides were not b(I). If they were consecutive, we could eliminate the first by substitution/elimination. Otherwise, in the AAB

```
b(I) = f1(I,J)
  .
  .
  .
```

```
      a(I) = f2(I,J)
           .
           .
           .
      b(I) = f3(I,J)
```

let i, j be any fixed values of I, J. By the restricted subscript property, any occurrence of a(i) in $f_1(i,j)$ must have subscript j. But the loop parameters guarantee that i > j so a(i) is not a reference to a member of the dependency set of b(I) $=f_1(I,J)$ and, again, we may combine the assignments to b(I) by substitution/elimination.

The first step in the transformation of the above nest is to unroll loop 10 to depth d. The result is

```
      DO 10 J = 1,d*(N/d)-d+1,d
         DO 20 I = J+1,M
            B(I) = F(I,J)
20       CONTINUE
         DO 30 I = J+2,M
            B(I) = F(I,J+1)
30       CONTINUE
            .
            .
            .
         DO 100 I = J+d,M
            B(I) = F(I,J+d-1)
100      CONTINUE
10    CONTINUE

C Following is the clean-up loop.

      DO 200 J = d*(N/d)+1,N
         DO 210 I = J+1,M
            B(I) = F(I,J)
210      CONTINUE
200   CONTINUE
```

We will ignore the clean-up loop in what follows since we do not transform it further. Consider loops 20 and 30 in the unrolled code. Write the sequence of statements resulting from substituting the first value of the DO variable in the range of loop 20 and adjusting the DO parameters of loop 20 accordingly. We refer to this operation as *peeling* the range of the loop for this value of the DO variable. Some paradigms involve peeling the range for several values of the DO variable at the beginning or end of its set of values. In this paradigm peeling is equivalent to writing special code to deal with the "triangle" formed at the top of a block of columns.

```
      B(J+1)=F(J+1,J)
      DO 20 I = J+2,M
            B(I) = F(I,J)
20    CONTINUE
```

```
      DO 30 I = J+2,M
            B(I) = F(I,J+1)
30    CONTINUE
```

Loops 20 and 30 are now parameter-equivalent. We wish to verify that these two loops satisfy the conditions of Theorem 2. Let k, c be allowable values of the DO parameter I, where $k < c$, and let j be an allowable value of the DO parameter J. Substituting these values, consider statements from the ranges of the indicated loops, as follows:

$b_s(c) = f_s(c,j)$ (loop 20)
$b_t(k) = f_t(k,j+1)$ (loop 30)

From the restricted subscript property, every occurrence of b_s in the expression f_t has subscript k or j+1. But this subscript is not c because $c > k$ and $c \geq j+2$; hence $b_s(c)$ is not a reference to a member of the dependency set of $b_t(k) = f_t(k,j+1)$. Similarly, every occurrence of b_t in the expression f_s has subscript c or j. But this subscript is not k because $k < c$ and $k \geq j+2$; hence $b_t(k)$ is not a reference to a member of the dependency set of $b_s(c) = f_s(c,j)$. By Theorem 2 the loops may be condensed resulting in

```
      B(J+1) = F(J+1,J)
      DO 40 I = J+2,M
         B(I) = F(I,J)
         B(I) = F(I,J+1)
40    CONTINUE
```

If $d > 2$, loop 40 is followed in the unrolled code by

```
      DO 50 I = J+3,M
         B(I) = F(I,J+2)
50    CONTINUE
```

We offer an inductive argument that the same transformations can be applied to this pattern of assignment statements and DO loops. Suppose that after n such steps the first n+1 inner loops have been mapped into the following:

```
C Following is the peeled code from the first step.
      B(J+1) = F(J+1,J)
C Following is the peeled code from the second step.
      B(J+2) = F(J+2,J)
      B(J+2) = F(J+2,J+1)
C The following dots represent the peeled code from
C steps 3 through n-1.
          .
          .
          .
```

```
C Following is the peeled code from the n^th step.
      B(J+n) = F(J+n,J)
      B(J+n) = F(J+n,J+1)
         .
         .
         .
      B(J+n) = F(J+n,J+n-1)

C Following is the DO loop resulting from condensing the
C first n-1 DO loops.
      DO 60 I = J+n+1,M
         B(I) = F(I,J)
            .
            .
            .
         B(I) = F(I,J+n)
60    CONTINUE
```

If $d > n+1$, loop 60 is followed in the unrolled code by

```
      DO 70 I = J+n+2,M
         B(I) = F(I,J+n+1)
70    CONTINUE
```

In loop 60 peel the range for the first value of the DO variable and adjust the parameters accordingly; then examine whether we may condense the new loop 60 and loop 70. The typical statements are

$b_s(c) = f_s(c, j+m)$ (new loop 60)
$b_t(k) = f_t(k, j+n+1)$ (loop 70)

where $0 \le m \le n$ and c, k, j are as before. Then every occurrence of b_s in f_t has subscript k or j+n+1. But $c > k$ and $c \ge j+n+2$, so such a subscript could not be c. Again, every occurrence of b_t in f_s must have subscript c or j+m. But $k < c$ and $k \ge j+n+2 \ge j+m+2$, so the subscript is not k. Hence the loops can be condensed by Theorem 2, completing the inductive argument. After d-1 steps all the inner loops have been condensed and we are ready to consider substitution/elimination. In the peeled code that precedes the condensed loop, successive definitions of $b_s(J+n)$ for $n \le d-1$ have the form

```
b_s(J+n) = f_s(J+n,J+m)
      .
      .
      .
b_s(J+n) =f_s(J+n,J+m+1)
```

where $0 \le m \le n-2$. The statements between the two definitions are assignments to $b_t(J+n)$ where $t \ne s$. Let j be any allowable value of J. From the restricted subscript property, every occurrence of b_t in the expression $f_s(j+n,j+m)$ must have subscript j+m; hence $b_t(j+n)$ cannot be a reference to a member of the dependency set of $b_s(j+n) = f_s(j+n,j+m)$. Therefore, we may

eliminate the first definition if we substitute $f_s(J+n,J+m)$ for $b_s(J+n)$ in all the statements up to and including the second definition. Repetition of the operation results in a series of AABs, one for each n, $1 \le n \le d-1$. In each AAB all the assignments to $b_s(J+n)$, for each s, have been collected into a single assignment.

Similar arguments about substitution/elimination obtain for the condensed loop where the sequence of definition statement, intervening statements, and redefinition statement looks like

```
b_s(I) = f_s(I, J+m)
          .
          .
          .
b_t(I) = f_t(I, J+m)  or  b_t(I) = f_t(I, J+m+1)
          .
          .
          .
b_s(I) = f_s(I, J+m+1)
```

where $0 \le m \le n-1$. Let i, j be any allowable values of I and J. Then any occurrence of b_t in $f_s(i,j+m)$ has subscript j+m by the restricted subscript property. However, the inner DO statement is DO <label> I = J+n+1,M and so $i \ge j+n+1 \ge j+m+2$. Hence, $b_t(i)$ is not a reference to a member of the dependency set of $b_s(i) = f_s(i,j+m)$ and the range of the DO may be compacted by substitution/elimination.

The final result of the transformations is Fortran of the following form where *B* is the same as in the original Fortran, i.e., the array names on the left side of statements in the blocks are the same as in the original block and appear in the same order.

```
      B(J+1) = G1(J, J+1)
      B(J+2) = G2(J, J+1, J+2)
          .
          .
          .
      B(J+d-1) = Gd-1(J, J+1, J+2, ..., J+d-1)
      DO 10 I = J+d,M
         B(I) = G(I, J, J+1, J+2, ..., J+d-1)
10    CONTINUE
```

The G blocks are derived from the F blocks by the above processes.

An example of a routine that fits this paradigm is F01AKY from [4]. This routine calculates $b = L^{-1}b$ where L has the form

$$\begin{matrix} L_{11} & 0 \\ L_{21} & I \end{matrix}$$

L_{11} is unit lower triangular of order N, and L_{21} is of order M-N by N. The original form of the code is as follows:

```
      SUBROUTINE F01AKY(A, IA, M, N, B)
      INTEGER IA, M, N
      DOUBLE PRECISION A(IA,N), B(M)
      INTEGER I, J
      DO 40 J=1,N
CDIR$ IVDEP
         DO 20 I=J+1,M
            B(I) = B(I)-A(I,J)*B(J)
   20    CONTINUE
   40 CONTINUE
         RETURN
         END
```

The comment CDIR$ IVDEP is a directive to CRAY compilers. It instructs the compiler to compile vector code even if its dependency analysis cannot verify that the code can be vectorized. The unrolling tool handles comments in such a way that such directives are preserved and are repeated in the clean-up loop.

The tools discussed in the next section were used to unroll this code to depth 4, condense, and substitute. The result is shown below. The comments generated by the tools assist in relating the transformed code to the transformations.

```
      SUBROUTINE F01AKY(A,IA,M,N,B)
      INTEGER IA,M,N
      DOUBLE PRECISION A(IA,N),B(M)
      INTEGER I,J
C*** DO-loop unrolled to depth 4 ***
      M99999 = (N- (1)+1)/ (4)
      M99998 = 1 + 4* (M99999-1)
      DO 20 J = 1,M99998,4
CDIR$ IVDEP
C*** DO loops condensed ***
         B(J+1) = B(J+1) - A(J+1,J)*B(J)
C*** DO loops condensed ***
C*** Redefinition detected - substitution/elimination applied ***
         B(J+2) = (B(J+2)-A(J+2,J)*B(J)) - A(J+2,J+1)*B(J+1)
C*** Redefinition detected - substitution/elimination applied ***
C*** Redefinition detected - substitution/elimination applied ***
         B(J+3) = ((B(J+3)-A(J+3,J)*B(J))-A(J+3,J+1)*B(J+1)) -
     +            A(J+3,J+2)*B(J+2)
         DO 10 I = (J+3) + 1,M
C*** Redefinition detected - substitution/elimination applied ***
C*** Redefinition detected - substitution/elimination applied ***
            B(I) = (((B(I)-A(I,J)*B(J))-A(I,J+1)*B(J+1))-A(I,J+2)*
     +              B(J+2)) - A(I,J+3)*B(J+3)
C*** Redefinition detected - substitution/elimination applied ***
   10    CONTINUE
   20 CONTINUE
```

```
      DO 40 J = M99998 + 4,N
CDIR$ IVDEP
            DO 30 I = J + 1,M
                  B(I) = B(I) - A(I,J)*B(J)
   30       CONTINUE
   40 CONTINUE
      RETURN
      END
```

Timing experiments show that on a CRAY-1S the original code executes on data with $N \geq 200$ at about 35 megaflops and the transformed code at about 65 megaflops, a speedup of over 80 percent.

6. The Transformation Tools

In this section we provide a general description of the tools that effect the transformations and certain Toolpack features on which they depend.

Toolpack[3] consists of a suite of software tools aimed at the development and maintenance of moderate-sized Fortran programs, a user interface to the tools, and a host-system interface called TIE (Tool Interface to the Environment). Both interfaces may vary from one host system to another. The tools are portable to any installation of TIE and are *integrated* in the sense that user-requested end results are obtained by invoking an appropriate sequence of tools. Each tool in the sequence takes its input from files generated by the user and/or other tools and it, in turn, generates output as files for the user and/or other tools. Scheduling the appropriate sequence of tool invocations and managing the intermediate data is the task of the user interface. The reader should refer to [2] and the documentation on the Toolpack distribution tape for detailed information on the installation and use of Toolpack tools, including those described here, and the software environment in which they reside.

Toolpack includes a Fortran lexical analyzer tool, sometimes called a "scanner," that maps a Fortran program into a sequence of basic lexical units called *tokens*. For example, the assignment statement

```
   20       C(I) = C(I) + A(I,5)*B(5)
```

may be represented as the following sequence of tokens:

[3]Toolpack is in the public domain and may be obtained from The National Energy Software Center, Argonne National Laboratory, Argonne, IL 60439, U.S.A. or The Numerical Algorithms Group, Inc., 1101 31st Street, Downers Grove, IL 60515, U.S.A. or The Numerical Algorithms Group, Ltd., 256 Banbury Road, Oxford OX2 7DE, U.K.

```
<integer constant> <name> <left parenthesis> <name>
        20           C                          I
<right parenthesis> <equals> <name> <left parenthesis>
                                    C
<name> <right parenthesis> <plus> <name> <left parenthesis>
   I                                  A
<name> <comma> <integer constant> <right parenthesis>
   I                    5
<star> <name> <left parenthesis> <integer constant>
          B                                  5
<right parenthesis> <end of statement>.
```

Some token types (e.g., <name> and <integer constant>) have strings associated with them; the associated string is shown under the token in the example. The Toolpack scanner outputs a file containing the sequence of tokens and their associated strings, except that the strings associated with tokens of type <comment> are stored in a separate file and indexed to their place in the token sequence. The two files together constitute the *token/comment stream.*

Another Toolpack tool, the parse tree builder or "parser," takes a token/comment stream as input and, by referring to a Fortran grammar, produces a representation of the program as a *parse tree,* composed of nodes and branches, together with an associated *symbol table.* To illustrate, the assignment statement above would be represented as a subtree of the program parse tree as shown in Fig. 1. The types of the nodes are shown in brackets. Nodes of certain types, such as [name] and [integer constant], have associated symbols in the symbol table. These are shown below the node.

Also of direct significance for the transformation tools is a Toolpack tool called "Polish" or the "unscanner" which maps a token/comment stream into Fortran text. User-defined options in an options file specify the rules for indentation, label increments, etc., that govern the appearance of the Fortran text. A Polish options editor tool facilitates the construction of options files.

The three representations of a Fortran program and the tools that map one into another are summarized in Fig. 2. (Toolpack names for the tools are shown in parentheses.)

Mapping a parse tree/symbol table into a token/comment stream is called "parse tree flattening" and is the type of mapping performed by Toolpack transformation tools such as the precision transformer and the declaration standardizer. Such tools call functions in the Toolpack system that enable the tool to extract information about the parse tree and that provide the facility to construct a token/comment stream by generating individually specified tokens and/or by generating the tokens corresponding to certain kinds of subtrees of the parse tree.

The unrolling, condensing, and substitution/elimination transformations are carried out by parse tree flattening tools. To supplement the Toolpack system functions we wrote additional parse tree flattening functions (largely by modifying existing functions) as utilities for these transformation tools.

The tools that carry out the transformations discussed in this paper are the DO loop unrolling tool (ISTUD), the DO loop condensing tool (ISTCD), and the substitution/elimination tool (ISTSB). These, like all Toolpack tools, are written in "pre-Fortran" that contains INCLUDE statements and symbolic constants; the tool source must be processed using a suitable macro processor, such as the portable Toolpack installation utility TIEMAC, to produce Fortran. These new tools conform to the requirements of the Toolpack virtual machine [2] and

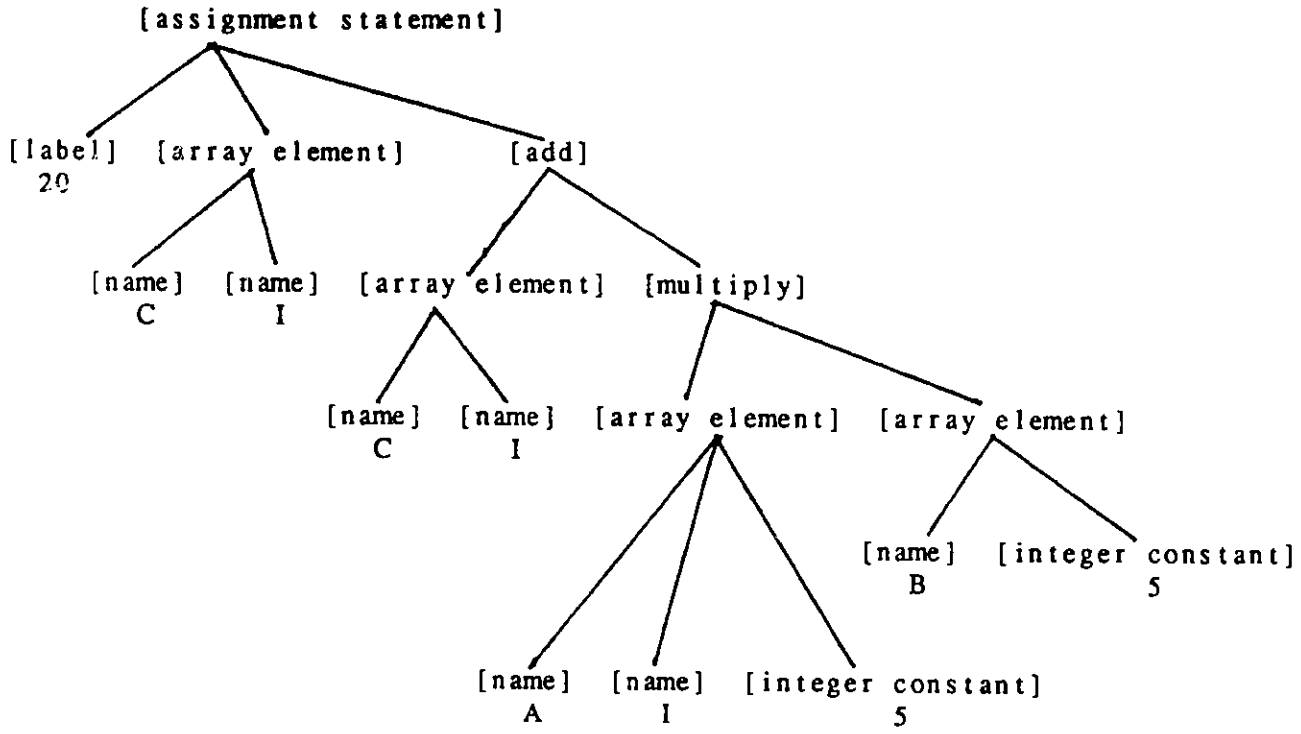

Figure 1. The Parse Tree Representation of
20 C(I) = C(I) + A(I,5)*B(5)

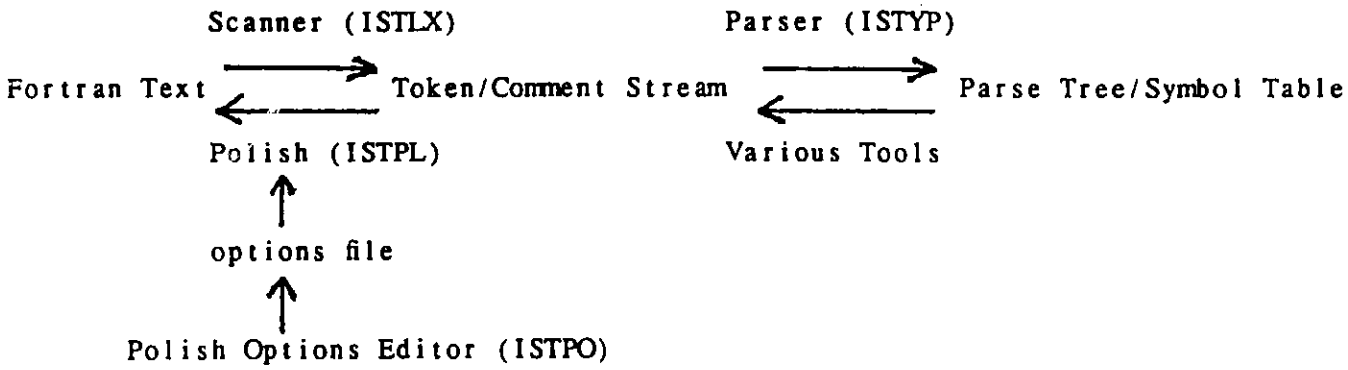

Figure 2. Transformations Among Fortran Representations

are designed to operate in the environment provided by Toolpack.

ISTUD searches a Fortran program (represented as a parse tree/symbol table) for DO loops, checking whether the loop is the outer DO of a nest and satisfies regularity conditions (1), (2), and (3). The tool unrolls such DO loops to a depth specified by the user. The transformed program is output as a token/comment stream.

ISTCD searches a Fortran program (represented as a parse tree/symbol table) for sequences of DO loops that match the patterns obtained by unrolling the outer loop of some one of a set of nested loop paradigms typically found in linear algebra routines. It condenses such sequences according to rules for the paradigm discovered (see [3]). The design of the tool encourages the incorporation of additional paradigms as experience dictates. The transformed program is output as a token/comment stream.

ISTSB searches sequences of assignment statements in a Fortran program (represented as a parse tree/symbol table) for opportunities to apply substitution/elimination. The strategy of ISTSB is sketched in [3] and includes permutation transformations when these are possible and result in an application of substitution/elimination. The sequences of greatest interest are those that result from the actions of ISTUD and ISTCD. The transformed program is output as a token/comment stream.

As noted in Section 4, the tools only check names in array references when examining membership in a dependency set. Formal analysis of the sort exemplified in Section 5 assures the user that the transformed program is correct if both the conditions that define an included paradigm, and the conditions described in warnings issued by the tools, are satisfied.

The flow of control among these tools is shown in Appendix B in the form of a "program" in an idealized language that permits nested *while* loops, invocation of tools, and the use of the termination status of a tool. Comments have "#" in column 1. Since ISTUD, ISTCD, and ISTSB are parse tree flattening tools, the parser, ISTYP, must be invoked after ISTUD to produce a parse tree/symbol table for input to ISTCD, and after ISTCD to produce a parse tree/symbol table for input to ISTSB.[4] Moreover, whenever ISTCD or ISTSB carries out a transformation, it sets the termination status to one of the flags *termflag_0* or *termflag_1* as explained in the comments. The termination status determines which further tool sequences are invoked. For example, we saw that each time two DO loops that match the paradigm of Section 5 are condensed there is the possibility that the new DO loop and its successor in the sequence will match the paradigm. Hence, whenever ISTCD carries out this transformation, it exits with the termination status set to *termflag_0*, which causes the parse tree/symbol table generated from the output token/comment stream to be submitted as input to ISTCD. For other paradigms, the sequence ISTYP/ISTSB/ISTYP/ISTCD is scheduled because ISTCD terminates with the status *termflag_1*. Again, whenever ISTSB carries out a substitution/elimination or permutation the termination status is set to *termflag_0*, causing the invocation of the sequence ISTYP/ISTSB. Thus these tools make full use of the integration of the Toolpack tool suite.

On a Unix installation of Toolpack, this flow of control is managed by a shell script that is logically equivalent to the program in Appendix B (neglecting error checking, file handling, etc.). The user types the command line

ucs <unrolling depth> <Polish options file> <Fortran source file>

[4]Although it is conceptually correct to say that ISTYP is "invoked," the design of Toolpack permits the parser tool to be called as a subroutine by the tools that require a parse tree. This is more efficient than invoking the parser as a free-standing tool.

The transform of the code in the file named <*Fortran source file*> is written to standard output and a log showing the sequence of tool invocations is written to standard error. The file named <Polish options file> is assumed to have been created using the Polish options editor; it governs the formatting of the transformed code by the unscanner ISTPL.

Normally, when ISTCD and ISTSB call for no further invocation sequences, the transformed program is mapped from a token/comment stream to Fortran text by ISTPL. However, if after ISTUD, ISTCD, and ISTSB have been invoked, no substitution/elimination transformations are possible, the presumption is that unrolling and (possibly) condensing have not served the purpose of producing Fortran better suited to a vectorizing compiler. *ucs* recognizes this condition and causes the program in its original form to be output.

Comments present in the original program are retained in the transformed code and are placed in "reasonable" locations, although there is no guarantee that they will be interposed optimally with the source code to which they relate. Handling comments takes on special significance because all compiler directives known to the authors take the form of special comments. We have already noted that ISTUD repeats comments in the clean-up loop. It is also necessary to be sure that the options used with ISTPL preserve comments in their original form so that their syntax as compiler directives is not changed.

7. Preliminary Results

We believe that this work can be viewed in at least two ways: as providing useful software tools for inclusion in the programming environment of a vector computer and as research into the means for developing such tools. The experimental results presented in this section should help evaluate the work from either orientation. We begin by outlining the criteria we have used to judge the tools and the work of developing them.

We first require, of course, that the tools shall reliably perform the task for which they have been programmed—they must produce software that is computationally equivalent to the original code, in the sense of producing the same output from the same input data. Our tests verify that this has been achieved for Fortran that matches any of the included paradigms, subject to conditions issued by the tools as warnings.

Secondly, we require that the transformed code be reasonably close in performance to that produced by human experts. The codes in our test bed satisfied this criterion, as shown below by the experimental results.

Thirdly, we require that the tools recognize and transform, at an acceptable cost, a sufficiently wide range of paradigms to be useful. Initially we have considered full linear algebra modules; these are at the heart of very many programs and have already been studied by experts. Our initial test bed has been a set of 21 kernels taken from the NAG library. These form a reasonably large, documented, and self-contained collection on which to work. Moreover, J. J. Du Croz of NAG had already studied these subroutines and unrolled them "by hand." The unrolling depth is 4 in both his work and the tests. The current versions of the tools recognize and transform 19 of the kernels. Paradigms to match the remaining two have not yet been incorporated.[5]

[5]One of these two kernels, namely f01agf, performs the reduction of a real, symmetric matrix to tridiagonal form. The matrix is held in compressed form (only the lower triangle). If one applies the tools to this code, ISTCD resorts to a general condensing algorithm and issues warnings as comments in the transformed code. The transformed f01agf performs the test calculations correctly but inefficiently. In particular, all the substitutions that are performed by ISTSB are contained in DO loops that are never executed. We expect to refine ISTCD to look for the "real" paradigm but we leave this example to illustrate the process through which the tools were refined. Data from a timing experiment using this code is included in Appendix A.

The cost of automatically transforming programs is expressible in terms of the cost of developing and maintaining the tools and the resources required to execute them. The source for ISTUD, ISTCD, and ISTSB, and their associated function library, currently comprises about 9000 lines of Fortran (prior to macro processing of the INCLUDE statements), including comments. The tools required about 8 man-months of effort to develop and document. The NAG kernels were transformed (unrolling depth 4 to permit comparison with Du Croz's work) on a VAX 11/780 at Argonne National Laboratory in about one and one-half hours of wall clock time when the machine load was low. We regard these costs as an acceptable price to pay for the benefits cited in Section 1.

We expect these tools to become more intelligent through the incorporation of additional paradigms and more sophisticated analysis. As the level of intelligence built into the system increases so also does the cost of using and maintaining the tools. We will need more experience before we can estimate the benefits and marginal cost of adding sophistication to the tools. We have been guided by the principle that the efficiency of the transformed Fortran weighs much more heavily than the running time or complexity of a tool.

The differences between the tool-transformed code and the human-transformed code mainly involve the clean-up loop. The tools use a single algorithm to create the clean-up loop while Du Croz used some alternative strategies that were beyond the scope of the tools. Since these differences do not appear critical for large N, we do not regard this as a significant area for refinement of the tools.

Our experiments involved executing and timing 25 test programs that call various members of the test bed of NAG kernels. This effort generated a plethora of information demonstrating the behavior of the code for various N between 50 and 200, the effect of memory bank conflicts, and the relative merits of different clean-up strategies.

All the results presented here were obtained on the CRAY-1S at AERE, Harwell. This is a single processor machine so the results are not confused by the effects of memory references from other CPUs. Furthermore, this machine has only one load/store pipe. The effect of one pipe may be important although initial experiments on other architectures suggest that the same effects are present.

Table 1 gives detailed performance figures for two user-callable test programs that perform matrix multiplication and the inversion of a real, symmetric, positive-definite matrix (using Cholesky factorization), respectively. The first column in the table is the dimension of the matrix. The versions of the NAG kernels called by the test program are indicated by ROLL, TOOL, and JJDC signifying, respectively, the original (or "rolled") version, the tool transformed version, and the version produced by Du Croz.

The cycle time of the CRAY-1S is 12.5 nanoseconds so that the maximum "vector" speed (which we define to be one result per cycle) corresponds to 80 megaflops. The simplest example of interest is that of matrix multiplication. The unmodified code reaches 39 megaflops while the unrolled codes both reach about 70 megaflops, a reasonable approximation to the peak "vector" performance. It has not been possible to achieve an average speed of more than one result per cycle ("super-vector" performance) from Fortran using these techniques. It is clear that for the longer vector lengths both adapted versions of the code show significant improvements (almost a doubling in speed) compared to the "rolled" code. Moreover, the differences between the automatically altered modules and those changed by Du Croz are very small.

Bank conflicts did not occur in obtaining the results in Table 1. However, other examples, where delays resulting from bank conflicts are present, show slower speeds but

qualitatively similar results; the rates achieved with the human and machine transformed modules are similar and show significant gains over the original code. In fact, the advantages are relatively greater because the memory references are more expensive, so their elimination is more effective.

Matrix multiplication is, in some respects, an artificial example because all of the floating-point arithmetic can be vectorized and the vector lengths are constant. The second example, Cholesky decomposition, contains varying vector lengths (the decomposition is triangular) and scalar, real arithmetic. This is reflected in the slower calculation rates shown in Table 1. Nonetheless, the comparative speed of the tool-changed code is much better than the speed of the unchanged code and close to the human-unrolled version. From the syntactic point of view this code is considerably more challenging than matrix multiplication.

In Appendix A we present a selection of data obtained from other user-callable test programs that call the kernels. A range of matrix decompositions, linear equation solution methods and eigenvalue problems is included, and several special forms are handled. The tools have succeeded according to the criteria outlined above. The first column in Appendix A is the size of N (dimension of the matrix), and "TOOL" and "JJDC" indicate that the test program called the tool-transformed or human-transformed (Du Croz) versions of the kernels, respectively. The entries are relative improvements, as percentages, over the original, "rolled" code. The thrust of our argument is that, with the exception noted, the tool-transformed kernels achieve the same improvements as the Du Croz versions. More than one column of results indicates either that runs with different sets of test data were made, to measure different options in the test program or, in the case of F01CKF, that A=BC, B=BC, and C=BC were calculated (sub-columns 1, 2, and 3, respectively). The second calculation is significantly affected by bank conflicts.

We observed some of the benefits of automation in the course of performing these experiments. In particular, it was relatively easy to process, or reprocess, the test bed when we made changes in the tools. This was important since refining the tools involves quite a lot of "cut and try." The implication is that when the computer environment alters, for example, a new release of a compiler, then one can easily repeat transformations and experiments. Moreover, it is possible that some small improvement can be gained which may be missed or judged not to repay the effort involved in rewriting code if one is doing it by hand. Our tests revealed several examples of this, not included in Appendix A, where we obtained gains not achieved by Du Croz.

8. Conclusions and Future Work

We have described a methodology and a set of software tools for improving the performance of Fortran subroutines on vector processors. Results of experiments performed on a CRAY-1S have been presented. These show that reasonable success has been achieved in terms of the speed of the transformed codes (in some cases speeds have been almost doubled) and their similarity to human-transformed codes. It has also been shown that Toolpack provides a suitable environment for the implementation of this type of software tool.

Future work related to these tools includes expanding the range of paradigms recognized, testing the unrolling technique on additional vector machines, and examining the effects of unrolling to different depths. Some further software engineering is planned, to improve the robustness of the tools.

The functions exemplified by these tools may be characterized as recognizing source code constructs that consume a significant amount of computing time and transforming them to

forms which are more suitable for a particular computer. In the longer term we intend to look at other architectures (for example, parallel machines with either global-shared memory or message-passing designs) with a view to recognizing source-code constructs that may be related to performance. For example, a segment of a program may be identified as a candidate for parallelization by tools that detect both large amounts of floating point arithmetic in the segment and data independence among subsegments. The components of ISTCD that determine and examine dependency sets provide a starting point for detecting data independence.

We anticipate that it will be important to be able to combine detailed information given by a tool that tests data dependence conditions with a programmer's global view of the program and the machine. It will also be important to easily conduct tests and experiments on variants of a program under development. Hence we envision that future tools will be interactive.

Acknowledgments

We wish to thank the Numerical Algorithms Group for permission to use the NAG kernels in our experiments, Jeremy Du Croz for his valuable counsel on transformation paradigms and timing experiments, Burton Garbow for his comments on the manuscript, and Gail Pieper for her assistance in preparing this manuscript.

References

[1] ANSI X3.9-1978. *Programming Language FORTRAN.* American National Standards Institute, Inc., New York, 1978.

[2] Cowell, W. R., Hague, S. J., and Iles, R. M. J. *Toolpack/1 Introductory Guide.* Publication NP1007, Numerical Algorithms Group, Downers Grove, Ill., and Oxford, U.K., 1985.

[3] Cowell, Wayne R., *The Toolpack Tools ISTUD, ISTCD, and ISTSB: Guide for Users and Installers.* MCS-TM-74, in preparation (also to be included on the Toolpack distribution tape).

[4] Daly, C. and Du Croz, J. J., Performance of a Subroutine Library on Vector Processing Machines. In *Vector and Parallel Processors in Computational Science,* ed. I. S. Duff and J. K. Reid, North-Holland, 1985.

[5] Dongarra, J. J. and Eisenstat, S. C. Squeezing the Most Out of an Algorithm in CRAY FORTRAN. *ACM Trans. Math. Software* 10 (1984): 221-230.

[6] Dongarra, J. J. and Hinds, A. R. Unrolling Loops in FORTRAN. *Software - Practice and Experience* 9 (1979): 219-226.

[7] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5 (1979): 308-323.

Appendix A. Percentage Decrease in Execution Time for Codes Unrolled to Depth 4

	TOOL	JJDC
ROUTINE: f01adf		
Inversion of real symmetric positive-definite matrix		
50	36	41
100	41	45
150	42	46
200	42	47
ROUTINE: f01aef		
Reduction of $Ax=\lambda Bx$ to standard symmetric eigenproblem		
50	37	40
100	42	44
150	43	45
200	44	45
ROUTINE: f01aff		
Back transformation of eigenvectors after f01aef (N vectors)		
50	40	46
100	43	47
150	43	47
200	44	47
ROUTINE: f01akf		
Reduction of real, unsymmetric matrix to Hessenberg form.		
50	33	34
100	39	39
150	40	41
200	42	42
ROUTINE: f01alf		
Back transformation of eigenvectors after f01akf (N vectors)		
50	23 45	15 29
100	27 44	21 34
150	31 46	24 36
200	32 45	27 38
ROUTINE: f01axf		
QR factorization (NxN)		
50	17	17
100	19	19
150	19	20
200	20	20

Appendix A. Percentage Decrease (continued)

	TOOL			JJDC		
ROUTINE: f01ckf						
Matrix multiplication $A=BC$						
50	45	48	44	46	49	46
100	46	44	46	46	44	46
150	46	48	45	46	48	45
200	46	32	45	45	32	45
ROUTINE: f01clf						
Matrix multiplication $A=BC^T$						
50	45			46		
100	46			46		
150	45			46		
200	45			45		
ROUTINE: f03aef						
LL^T factorization						
50	32			34		
100	38			40		
150	41			42		
200	42			43		
ROUTINE: f03aff						
LU factorization (real)						
50	23			26		
100	30			34		
150	33			38		
200	35			40		
ROUTINE: f04agf						
Solution of $LL^Tx=b$						
50	40	40		41	40	
100	43	43		43	42	
150	43	43		43	43	
200	44	44		44	43	
ROUTINE: f04ajf						
Solution of $LUx=b$ (real)						
50	28	41		32	46	
100	33	44		36	48	
150	34	44		38	48	
200	36	44		40	48	

Appendix A. Percentage Decrease (continued)

	TOOL	JJDC
ROUTINE: f04anf		
Solution of $QRx=b$ (NxN)		
50	14	14
100	14	14
150	13	13
200	13	13
ROUTINE: f01agf (The noted exception - see footnote 5.):		
Reduction of a real, symmetric matrix to tridiagonal form		
50	-71	16
100	-58	17
150	-48	17
200	-40	17

Distribution for ANL-85-63

Internal:

J. M. Beumer (2)
W. R. Cowell (80)
K. L. Kliewer
A. B. Krisciunas
P. C. Messina
G. W. Pieper
B. A. Simmons

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (168)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
  J. L. Bona, U. Chicago
  T. L. Brown, U. of Illinois, Urbana
  P. Concus, LBL
  S. Gerhart, MCC, Austin, TX
  G. H. Golub, Stanford U.
  W. C. Lynch, Xerox Corp., Palo Alto
  J. A. Nohel, U. of Wisconsin, Madison
D. Austin, ER-DOE
J. Greenberg, ER-DOE
G. Michael, LLL

Appendix B. Flow of Control for Transformation Tools

```
# Represent the original Fortran as a token/comment stream.
invoke ISTLX

# Represent the original Fortran as a parse tree/symbol table.
# See footnote 6.
invoke ISTYP

# BEGIN PHASE 1 - Unrolling
# ISTUD input is the original program as a parse tree/symbol table.
# Unroll and parse (parse tree/symbol table -> token/comment stream
#                          -> parse tree/symbol table)
invoke ISTUD
invoke ISTYP

# BEGIN PHASE 2 - Condensation
# Condense DO loops when paradigms are discovered.
# (parse tree/symbol table -> token/comment stream)
invoke ISTCD

# ISTCD sets the termination status to 'termflag_1' when the
# paradigm requires substitution/elimination before further
# condensation.  When any other paradigm is discovered, ISTCD
# sets the termination status to 'termflag_0'.

while (termination status = termflag_1)
#       Parse the output from ISTCD.
        invoke ISTYP

#       Apply substitution/elimination in preparation for further
#       search by ISTCD for paradigms.
#       (parse tree/symbol table -> token/comment stream)
        invoke ISTSB

#       ISTSB sets the termination status to 'termflag_0' whenever
#       any transformation is made.

        while (termination status = termflag_0)
#               Parse the output from ISTSB and invoke ISTSB again.
                invoke ISTYP
                invoke ISTSB
end while
```

Appendix B. Flow of Control (continued)

```
#       Parse the output from ISTSB and invoke ISTCD.
        invoke ISTYP
        invoke ISTCD
end while

while (termination status = termflag_0)
#       Parse the output from ISTCD and invoke ISTCD again.
        invoke ISTYP
        invoke ISTCD
end while


#       Parse the output from ISTCD
        invoke ISTYP

# BEGIN PHASE 3 - Substitution/Elimination
        invoke ISTSB

        while (termination status = termflag_0)
#       Parse the output from ISTSB and invoke ISTSB again.
        invoke ISTYP
        invoke ISTSB
end while

# "Unscan" the transformed token/comment stream.
# The output from ISTPL is the transformed Fortran.
        invoke ISTPL
```