SPECIAL ISSUE

Articles

ADVANCED COMPILER OPTIMIZATIONS FOR SUPERCOMPUTERS

Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.

DAVID A. PADUA and MICHAEL J. WOLFE

Supercomputers[1] use parallelism to provide users with increased computational power. Most supercomputers are programmed in some higher level language, commonly Fortran; all supercomputer vendors provide Fortran compilers that detect parallelism and generate parallel code to take advantage of the architecture of their machines[2] [25, 46, 53].

This article discusses some of the common (and not so common) features that compilers for vector or multiprocessor computers must have in order to successfully generate parallel code. The many examples given throughout are related to the generic types of machines to which they apply. Where appropriate, we also relate these parallel compiler optimizations to those used in standard compilers.

Fortran is currently the programming language of choice for supercomputers, largely because vendors presently provide optimizing, vectorizing, and concurrentizing[3] compilers only for Fortran. In addition, Fortran has historically been the most frequently used numerical programming language, and much investment has been made in its programs. The examples in the text are written in Fortran, using some of the new features described in the latest Fortran-8x proposal [7], such as the `end do` statement and array assignments. The literature distinguishes two types of concurrent loops: `doall` and `doacross` [16, 43]. The latter imposes a partial execution order across iterations in the sense that some of the iterations are forced to wait for the execution of some of the instructions from previous iterations. `Doall` loops do not impose any partial ordering across iterations, even though there may be critical regions in the loop bodies.[4] Despite our use of Fortran, none of the features or transformations explained here are peculiar to this language, and they can be successfully applied to many others.

[1] Some of the supercomputers available today are the FX Series from Alliant Computer Corporation, the Cyber 205 from Control Data Corporation, the Convex C-1 from Convex Computer Corporation, the Cray 2 and the Cray X-MP [15] from Cray Research, the Facom VP from Fujitsu [39], the S-810 Vector Processor from Hitachi [40], the SX System from NEC Corporation, and the SCS-40 from Scientific Computer Systems. Alliant's FX and Convex's C-1 are usually classified as minisupercomputers.

[2] Besides the vendor-supplied compilers, there are a number of experimental and third-party source-to-source restructurers. Among them are the University of Illinois's Parafrase [32], KAI's KAP [17, 18], Rice University's PFC [5], and Pacific Sierra's VAST [11].

This work was supported in part by the National Science Foundation under Grant US NSF DCR84-06916 and US NSF DCR84-10110, the U.S. Department of Energy under Grant US DOE-DE-FG02-85ER25001, and the IBM Donation.

[3] The term *parallelize* is often used to describe the translation of serial code for parallel computers. We prefer the more specific *concurrent* over parallel, since parallel may mean vector, concurrent, or lockstep multiprocessor computation. Also, Alliant Computer Corporation, the first commercial vendor to supply a compiler that automatically translates code for multiple processors, uses the term *concurrent*.

[4] The DOALL statement was originally defined in the Burrough's FMP Fortran [13, 36].

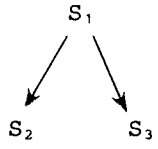
We will discuss how data-dependence testing of some form is required in any compiler that wishes to detect parallelism in serial code, and explain the different types of parallel code that can be generated. Vector code is appropriate for computers with vector instructions sets, while concurrent constructs are used in multiprocessor environments. Also covered are ways to improve the computation of the data-dependence graph. A (very incomplete) catalog of transformations and restructuring tricks that compilers use to optimize code for parallel computers is given, followed by a discussion of how these compilers communicate with programmers.

DATA DEPENDENCE

Data-dependence testing [5, 8, 10, 52] is required for any form of automatic parallelism detection. Data-dependence relations are used to determine when two operations, statements, or two iterations of a loop can be executed in parallel. For instance, in the code

```
S1: A = B + C
S2: D = A + 2.
S3: E = A * 3.
```

statements S_1 and S_2 cannot be executed at the same time since S_2 uses the value of A that is computed by S_1. This is called *true dependence* or *flow dependence* since the data value flows from S_1 to S_2, and is denoted $S_1\ \delta\ S_2$. S_3 also depends on S_1 (denoted $S_1\ \delta\ S_3$); thus S_1 must be executed before both S_2 and S_3. The data-dependence relations are often depicted in a data-dependence graph, with arcs representing the relations, as follows:



Notice that S_2 and S_3 are not connected by data-dependence arcs and so may be executed in parallel if two processors are available.

Two other kinds of data dependence are important. In the program segment

```
S1: A = B + C
S2: B = D / 2
```

S_1 uses the value of B before S_2 assigns a new value to B. Since S_1 is to use the "old" value of B, it must be executed before S_2; this is called *antidependence*, as the relation is from the use to the assignment, and is denoted $S_1\ \bar{\delta}\ S_2$. The third kind of dependence is shown in the program segment below:

```
S1: A = B + C
S2: D = A + 2
S3: A = E + F
```

Here S_3 assigns a new value to A after S_1 has already given a value to A. If S_1 is executed after S_3, then A will contain the wrong value after this program segment. Thus, S_1 must precede S_3; this is called *output dependence* and is denoted $S_1\ \delta^\circ\ S_3$.

The flow of control must also be taken into account when building data-dependence relations. For instance, in the program segment

```
S1: A = B + C
    if ( X >= 0 ) then
S2:    A = 0.
    else
S3:    D = A
    end if
```

the relations $S_1\ \delta^\circ\ S_2$ and $S_1\ \delta\ S_3$ hold, but $S_2\ \delta\ S_3$ does not hold even though S_2 assigns a value to A and S_3 uses A, and S_3 appears after S_2 in the program. Since S_2 and S_3 are on different branches of the same `if` statement, the value of A used in S_3 will never come from S_2.

Since the actual execution flow of a program is not known until run time, a data-dependence relation does not always imply data communication or memory conflict. For instance, in this program segment

```
S1: A = B + C
C1: if ( X >= 0 ) then
S2:    A = A + 2
    end if
S3: D = A * 2.1
```

the data-dependence relations $S_1\ \delta\ S_3$ and $S_2\ \delta\ S_3$ will both be computed by the compiler, even though S_3 will in fact take the value of A from only one of S_1 or S_2, depending on the value of X.

Many compilers also use the concept of dependence from an `if` statement to the statements under control of the `if`. This is called *control dependence* and is denoted δ^c. In the program segment above, for example, the control-dependence relation $C_1\ \delta^c\ S_2$ holds. Control-dependence relations are often added to the data-dependence graph in order to find which statements in the program can be reordered, or when looking for cycles in the graph.

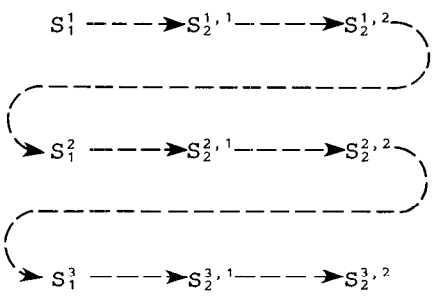
Data Dependence in Loops

Inside loops we are interested in data-dependence relations between statements and also dependence

relations between instances of statements. We distinguish between different instances of execution of a statement by superscripting the statement label with the loop iterations. For instance, in the loop

```
      do I = 1,3
S1:      A(I) = B(I)
         do J = 1,2
S2:         C(I,J) = A(I) + B(J)
         end do
      end do
```

statement S_1 is executed three times: S_1^1, S_1^2, S_1^3; and S_2 is executed six times: $S_2^{1,1}$, $S_2^{1,2}$, $S_2^{2,1}$, $S_2^{2,2}$, $S_2^{3,1}$, $S_2^{3,2}$ (we put the outer loop iteration number first). We can also draw the iterations as points with Cartesian coordinates, as below:



This diagram illustrates the iteration space; the dotted arrows show the order in which the instances of the statements are executed.

To find data-dependence relations in loops, the arrays and subscripts are examined. In the loop

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I)
      end do
```

the relation $S_1\ \delta\ S_2$ holds, since, for any iteration i, S_1^i will assign A(i) and S_2^i will use A(i) on the same iteration of the loop. Since the dependence stays in the same iteration of the loop, we say $S_1\ \delta_{=}\ S_2$.

In the following similar loop

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I-1)
      end do
```

the relation $S_1\ \delta\ S_2$ still holds, but for any iteration i, S_2^i will use an element of A that was assigned on the previous iteration of the loop by S_1^{i-1} (except S_2^2, which uses an "old" value of A(1)). Since the dependence flows from iteration $i-1$ to iteration i, we say that $S_1\ \delta_{<}\ S_2$ (the relation is $\delta_{<}$ because $i - 1 < i$).

A third similar example is shown below:

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I+1)
      end do
```

In this loop, for any iteration i, S_2^i will use an element of A that will be reassigned by S_1^{i+1}. Since S_2 should use an "old" value of A, the antidependence relation $S_2\ \overline{\delta}\ S_1$ holds. This relation flows from iteration i to iteration $i+1$, so we say $S_2\ \overline{\delta}_{<}\ S_1$ (since $i < i + 1$).

The = or < used as the subscript of δ is called the *data-dependence direction*, since it gives the direction of the dependence relation in the iteration space. In nested loops, there is a direction for each loop; these comprise a data-dependence direction vector. For instance, in the loop

```
      do I = 1,N
         do J = 2,N
S1:         A(I,J) = A(I,J-1) + B(I,J)
S2:         C(I,J) = A(I,J) + D(I+1,J)
S3:         D(I,J) = 0.1
         end do
      end do
```

the following dependence relations hold:

$S_1\ \delta_{=,<}\ S_1$
$S_1\ \delta_{=,=}\ S_2$
$S_2\ \overline{\delta}_{<,=}\ S_3$

Since most array subscripts are simple, simple tests are usually sufficient. Some compilers (e.g., Cray Fortran (CFT) and Control Data's Fortran 200 compilers) use restricted tests that allow only certain array subscript expressions, such as A(I) or A(I*c+k) (where c and k are constants). Dependence is assumed for any array reference that does not conform to the restrictions. These tests work well for new codes that are written with a particular computer in mind, since programmers will know what loops they want executed in parallel and will help the compiler by keeping their loops simple.

Sophisticated methods have been developed to handle more general cases (see [5, 8, 10, 52]). For instance, in order for data flow-dependence to be caused by the two array references to A here

```
      do I = L,U
S1:      A(c*I+j) = ...
S2:      ... = A(d*I+k)
      end do
```

(where c, d, j, and k are integer constants), the greatest common divisor of c and d (GCD (c,d)) must divide $(k-j)$. For example, no dependence would exist in the following loop

```
    do I = L,U
S1:     A(2*I) = ...
S2:     ... = A(2*I+1)
    end do
```

since the GCD $(2,2)=2$, which does not divide $1-0=1$.

More importantly, there must exist two values of the loop index variable I, say x and y, such that

$$L \leq x \leq y \leq U$$
$$c*x+j = d*y+k$$

In the sample loop below

```
    do I = 1,10
S1:     A(19*I+3) = ...
S2:     ... = A(2*I+21)
    end do
```

the only two values that satisfy this dependence equation are $x = 2$ and $y = 10$:

$$19*2+3 = 41 = 2*10+21$$

Solving this Diophantine equation is the subject of several of the references cited above, which also generalize this to nested loops where several loop indexes appear in a single subscript.

CODE GENERATION

When a good data-dependence graph has been built for a loop nest, the compiler can generate parallel code. Since many supercomputers have vector instruction sets, vectorization is important. Most vector computers can compute certain reduction operations, such as the sum of all the elements of a vector, using vector instructions. At least one supercomputer also has hardware to assist in the solution of first order linear recurrences.[5]

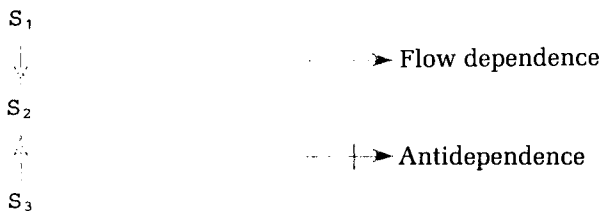
Newer supercomputers achieve higher speeds by using multiple processors. For these machines, generation of concurrent code will utilize all processors. Concurrent loops and concurrent blocks of code are two types of parallel code that can be generated.

Loop Vectorization

Loops are vectorized by examining the data-dependence graphs for innermost loops. A simple graph algorithm to find cycles in the data-dependence graph identifies any trouble spots in vectorization. If there are no cycles in the graph, then the whole loop can be vectorized. For example, the following loop

```
    do I = 1,N
S1:     A(I) = B(I)
S2:     C(I) = A(I) + B(I)
S3:     E(I) = C(I+1)
    end do
```

has the following data-dependence graph:



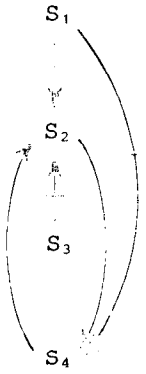
Because the data-dependence graph has no cycles, it can be completely vectorized, although some statement reordering will be necessary. Since $S_3 \ \overline{\delta} \ S_2$, S_3 must precede S_2 in the vectorized loop:

```
S1: A(1:N) = B(1:N)
S3: E(1:N) = C(2:N+1)
S2: C(1:N) = A(1:N) + B(1:N)
```

The following loop contains a data-dependence cycle:

```
    do I = 2,N
S1:     A(I) = B(I)
S2:     C(I) = A(I) + B(I-1)
S3:     E(I) = C(I+1)
S4:     B(I) = C(I) + 2.
    end do
```

The following is the data-dependence graph for this loop:



Statements S_2 and S_4 comprise a cycle in the data-dependence graph; when the dependence graph

[5] A first order linear recurrence is defined by the equations $x_i = c_i + a_i \times x_{i-1}$ $i = 2, 3, \ldots, n$ and $x_1 = 0$.

contains a cycle, the strongly connected components (or maximal cycles) must be found. All the statements in a data-dependence cycle must be executed in a serial loop unless the cycle can be broken. However, other statements may still be vectorized. Thus, the previous loop may be partially vectorized as follows:

```
S1: A(2:N) = B(2:N)
S3: E(2:N) = C(3:N+1)
    do I = 2,N
S2:    C(I) = A(I) + B(I-1)
S4:    B(I) = C(I) + 2.
    end do
```

A sufficient (but not necessary) condition for vectorization of a loop is that no upward data-dependence relations ($S_i \ \delta \ S_j$, and S_j lexically precedes S_i) appear in the loop. This will guarantee legal vectorization, but will miss loops where simple statement reordering would allow vectorization (as in the first example above).

Certain reduction operations appear frequently in programs and are recognized by vectorizing compilers. Prominent among these is the SUM of a vector:

```
    do I = 1,N
S1:    A(I) = B(I) + C(I)
S2:    ASUM = ASUM + A(I)
    end do
```

Vector code for this loop would appear as follows:

```
S1: A(1:N) = B(1:N) + C(1:N)
S2: ASUM = ASUM + SUM(A(1:N))
```

Here, the SUM function returns the sum of its argument. A special case of a SUM is a dot product; this is important because many supercomputers have a distinct adder and multiplier that can perform a dot product (a SUM of a multiplication) in the same time as a SUM alone, thus performing the multiplication with almost no time penalty.

Care must be taken by the compiler and the programmer to ensure that the correct answer will always result. Most methods to produce a sum using vector instructions involve accumulating partial sums, then adding the partial sums together. Since this will add the arguments in a different order than the original loop, round-off errors may accumulate differently, and some programs may produce substantially different answers. Some compilers have a switch that will disable generation of reductions in order to guarantee the same answers from the vectorized code as from the serial code.

Other common reductions are the PRODUCT of a vector or maximum or minimum of a vector. Simple pattern recognition can be used to find these operations (as well as SUM) in loops. More complicated patterns, such as a loop that finds the maximum of a vector and saves the index of the maximum, are also frequently found:

```
IMAX = 1
AMAX = A(1)
do I = 2,N
   if ( A(I) > AMAX ) then
      AMAX = A(I)
      IMAX = I
   end if
end do
```

Recognizing and generating vector code for such multistatement patterns enhance the overall power of a vectorizer.

At least one supercomputer has been designed with an instruction to solve first order linear recurrences.[6] These recurrences can be recognized by pattern matching:

```
do J = 2,N
   A(J) = A(J-1) * B(J) + C(J)
end do
```

Fast algorithms for solving recurrences on parallel computers have been devised that may be useful for some systems, although these suffer from the same round-off error accumulation problem mentioned earlier. Many computer systems offer a library of procedures to compute certain common forms of recurrences;[7] some compilers recognize these recurrences and translate them into calls to the appropriate library procedure.

Loop Concurrentization

One way to use multiple processors in a computer is to partition the set of iterations of a loop and assign a different subset to each processor [16, 19, 34, 38, 42, 43]. Two important factors that determine the quality of the concurrent code are the balance of processor load, and the amount of processor idle time due to synchronization. Concurrentization, therefore, should aim at an even distribution of the iterations among processors and should try to organize the code so as to avoid synchronization or, at least, to minimize waiting.

Checking for independent iterations is done by examining the data-dependence directions. If all the data-dependence relations in a loop have an = direction for that loop, then the iterations are independent. The only time that communication between

[6] The Burroughs Scientific Processor (BSP) [30] was designed with a recurrence instruction, but the project was canceled before the first machine was delivered. Among current supercomputers, the Hitachi S-810 has a recurrence instruction.

[7] In particular, the STACKLIB library for the Control Data 6600 [48], 7600, Cyber 70, Cyber 170, and Cyber 205 computers.

processors is necessary is when a data-dependence relation exists in the loop with a $<$ direction for that loop. For instance, the loop

```
    do I = 1,N
       do J = 2,N
S1:       A(I,J) = B(I,J) + C(I,J)
S2:       C(I,J) = D(I,J) / 2
S3:       E(I,J) = A(I,J-1)**2
                   + E(I,J-1)
       end do
    end do
```

has the following data dependences:

$S_1\ \delta_{=,<}\ S_3$
$S_1\ \overline{\delta}_{=,=}\ S_2$
$S_3\ \delta_{=,<}\ S_3$

Since all the data-dependence directions for the I loop are $=$, each iteration of the I loop can be executed in parallel. If N processors are available, each processor can execute one iteration of the loop. If fewer processors are available, the iterations can be folded onto the processors in one of several ways. The compiler can preschedule the iterations of the loop onto the P processors either in contiguous blocks

processor 1 executes iterations 1, 2, ..., $\lceil N/P \rceil$
processor 2 executes iterations $\lceil N/P \rceil + 1$, ... $2\lceil N/P \rceil$
:

or by assigning every Pth iteration to the same processor:

processor 1 executes iterations 1, $P+1$, $2P+1$, ...
processor 2 executes iterations 2, $P+2$, $2P+2$, ...
:

Alternatively, the processors can be self-scheduled [27, 47], meaning that each processor at the end of every iteration enters a critical section of code to determine what iteration of the loop it should execute next. Self-scheduling works well when the workload for each iteration is relatively large, but may vary between different iterations, perhaps due to conditional code in the loop.

Sometimes the iterations of a loop are not independent:

```
    do I = 2,N
S1:    A(I) = B(I) + C(I)
S2:    C(I) = D(I) * 2.
S3:    E(I) = C(I) + A(I-1)
    end do
```

The data-dependence relations for this loop are the following:

$S_1\ \overline{\delta}_{=}\ S_2$
$S_1\ \delta_{<}\ S_3$
$S_2\ \delta_{=}\ S_3$

Since there is a $<$ data-dependence direction for this loop, the iterations cannot be executed independently. If the different iterations are to be executed in parallel, the processor executing iteration i must not fetch the value for A($i-1$) in statement S_3^i before the processor executing iteration $i-1$ has stored the value of A($i-1$) in statement S_1^{i-1}. The code inside the loop with the synchronization added would appear as follows:

```
S1: A(I) = B(I) + C(I)
    signal ( I )
S2: C(I) = D(I) * 2.
    if ( I > 2 ) wait ( I-1 )
S3: E(I) = C(I) + A(I-1)
```

The compiler can sometimes reorder statements to reduce the effect of the required synchronizations. Weak data-dependence tests may add some synchronizations that are not really necessary, so good dependence testing is critical for good performance.

Recognition of simple reductions applies to concurrent loops as well as vector loops. One method to generate parallel code for the loop

```
    do I = 1,N
S1:    A(I) = B(I) + C(I)
S2:    D(I) = A(I) * 2.
S3:    ASUM = ASUM + A(I)
    end do
```

is to add synchronization (as above) around statement S_3, so that each processor p (of the P available processors) would execute the following loop:

```
    do I = p,N,P
S1:    A(I) = B(I) + C(I)
S2:    D(I) = A(I) * 2.
       if( I > 1 ) wait (I-1)
S3:    ASUM = ASUM + A(I)
       signal (I)
    end do
```

This method is simple to implement and will always result in the same answer as the original scalar code (which, again, can be an important factor due to round-off error accumulations). A faster method is to accumulate partial sums on each processor

```
    ASUMX(p) = 0
    do I = p,N,P
S1:    A(I) = B(I) + C(I)
S2:    D(I) = A(I) * 2.
S3:    ASUMX(p) = ASUMX(p) + A(I)
    end do
```

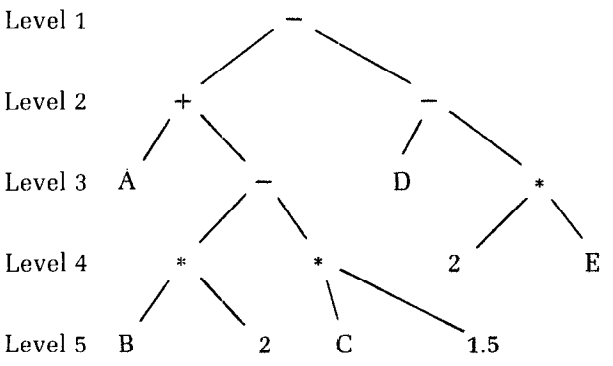
and then add the partial sums into the summation variable ASUM at the end of the loop. This produces completely parallel code without synchronization, but may accumulate different round-off errors.

High-Level Spreading

Another approach to utilizing multiple processors is to spread independent operations over several processors. A fine-grain parallelism model would spread the computation tree of an expression evaluation over several processors. For instance, the computation tree for the expression

```
A + (B*2 - C*1.5) - (D - 2*E)
```

is as follows:



In the computation tree at levels 2, 3, and 4, there are two arithmetic operations that could be performed simultaneously. With two processors, this expression could be computed in four times steps (assuming + and * take the same time and assuming no time for variable fetching or processor communication), instead of the seven time steps necessary for a single processor.

The problem with this model is that the assumptions are invalid for current machines; variable fetching does take time, and if several processors share a common memory, they can interfere with each other. Communication between processors also takes time; thus, it is better to try to spread blocks of code that represent a relatively large workload. As with concurrent loops, it is best if the blocks of code assigned to different processors are completely independent, so that no communication costs are incurred. If the workload is spread unevenly over the processors, then some processors may finish early and be left idle while other processors are still busy.

Adjacent blocks of code, such as adjacent independent recurrence loops or procedure calls, are candidates for high-level spreading [50]. The data-dependence relations between these high-level objects are examined, and if the blocks of code are independent, then they can be spread over several processors. If there are data-dependence relations between the blocks of code, then either synchronization must be added (as in loop concurrentization) to perform spreading, or the code must be executed serially.

The potential for speedup with spreading is much lower than for loop concurrentization. Loop concurrentization may find N independent blocks of code, where N is the loop bound. Spreading, in practice, will usually find only 2–3 independent blocks of code suitable for parallel execution.

Trade-offs between Vectorization and Concurrentization

Some recent computer designs (e.g., the Cray X-MP, Alliant FX/8, and ETA[10]) have multiple processors with vector instructions. The techniques of vectorization and concurrentization must be used together to take full advantage of these computers. When only one loop exhibits any parallelism, the compiler must decide whether to generate vector code, concurrent code, or whether to split the index set into segments, all of which can be executed concurrently and in vector mode. When the loop contains many `if` statements that would produce sparse vector operations, concurrent execution may be more efficient.

When several nested loops exhibit parallelism, the compiler must choose which loop to execute in vector mode and which in concurrent mode. Several factors should be considered. For example, since the vector speed of some machines depends on the stride of memory accesses, the compiler may choose to execute the loop that generates stride-1[8] memory operations in vector mode and some other loop in concurrent mode. However, since a < data-dependence direction implies that a synchronization would be required for concurrent execution, the compiler may attempt to execute a loop with a < direction in vector mode, and to choose a loop with all = directions for concurrent execution. This trade-off is illustrated in the sample loop below:

```
      do J = 1,N
         do I = 1,N
S1:         A(I,J+1) = B(I,J) + C(I,J)
S2:         D(I,J) = A(I,J) * 2.
         end do
      end do
```

This loop can be compiled with the I loop in vector mode, which will generate stride-1 memory operations (assuming Fortran column-major storage

[8] By stride-1 we mean vector operations where the vector elements are stored in consecutive memory locations.

order), and with the J loop in concurrent mode, as follows:

```
    doacross J = 1,N
S1:     A(1:N,J+1) = B(1:N,J)
                   + C(1:N,J)
        signal ( J )
        if ( J > 1 ) wait ( J-1 )
S2:     D(1:N,J) = A(1:N,J) * 2.
    end doacross
```

This may produce the best vector execution speed, but the data-dependence relation $S_1 \delta_< S_2$ requires synchronization in the concurrent loop. An alternate method to compile the loop would perform the J loop in vector mode:

```
    doall I = 1,N
S1:     A(I,2:N+1) = B(I,1:N)
          + C(I,1:N)
S2:     D(I,1:N) = A(I,1:N) * 2.
    end doall
```

Although this loop would have better concurrent execution speed, it would perhaps be at the expense of slower vector execution. Balancing the different methods of compiling the loop to get the best performance is a tough job for the compiler.

IMPROVING POTENTIAL PARALLELISM

The previous sections should clarify the importance of a good data-dependence testing procedure. If unnecessary relations are added to the data-dependence graph, then the potential for parallelism discovery can be reduced dramatically. Some methods for computing a more accurate data-dependence graph are given here.

Induction Variable Recognition

Variables in loops whose successive values form an arithmetic progression are called *induction variables*; the most obvious example of an induction variable is the index variable of a loop. Induction variables are often used in array subscript expressions. Traditional optimization techniques are aimed at finding induction variables to remove them from the loop and also to optimize the array address calculation [2]. For data-dependence tests, the array subscripts should be known in terms of the loop index variables; therefore, discovery of induction variables is important. Most compilers will recognize that the loop

```
INC = N
do I=1,N
   I2 = 2*I-1
   X(INC) = Y(I) + Z(I2)
   INC = INC - 1
end do
```

refers to the same array addresses as the following loop:

```
do I=1,N
   X(N-I+1) = Y(I) + Z(2*I-1)
end do
```

The expression assigned to I2 is recognized as a function of the loop index variable, so I2 is easily recognized as an induction variable. The assignment to INC is a self-decrement, which qualifies INC as an induction variable. If the last values of I2 and INC are not used later in the program, the two loops above may be used interchangeably. After vectorization, both loops become

```
X(N:1:-1) = Y(1:N) + Z(1:2*N-1:2)
```

Wraparound Variable Recognition

Sometimes a variable may look like an induction variable, but does not quite qualify. The assignment to J in the loop

```
J = N
do I=1,N
   B(I) = (A(J) + A(I)) / 2.
   J = I
end do
```

appears to qualify J as an induction variable, but J is used before it is assigned. In fact, the programmer used a trick to make the array A look like a cylinder. The loop takes the average of two adjacent elements of the array A; in the first iteration, the neighbor of A(1) is defined to be A(N)—the J variable accomplishes this trick. J is called a *wraparound variable*, since the values assigned to it are not used until the next iteration of the loop.

By peeling off one iteration of the loop, J can be treated as a normal induction variable:

```
if (N >= 1) then
   B(1) = (A(J) + A(I)) / 2.
   do I=2,N
      B(I) = (A(I-1) + A(I)) / 2.
   end do
end if
```

The if is necessary to test the zero-trip condition of the loop. The loop may be vectorized to become

```
if (N >= 1) then
   B(1) = (A(J) + A(I)) / 2.
   B(2:N) = (A(1:N-1) + A(2:N)) / 2.
end if
```

Symbolic Data-Dependence Testing

As mentioned in the data-dependence section, the simplest data-dependence subscript tests will be sufficient for a large number of cases, but are too lim-

ited for a general-purpose powerful compiler. Even some of the more sophisticated tests have severe restrictions, such as requiring that the loop bounds be compile-time constants. To handle a large majority of cases, a compiler must be able to compute precise data-dependence relations for very general array references. For instance, in the following loop

```
    do I = LOW,IGH
S1:     A(I) = B(I) + C(I)
S2:     D(I) = A(I-1)
    end do
```

the data-dependence relation $S_1\ \delta_<\ S_2$ can be computed using the simplest tests. However, in the similar loop

```
    do I = LOW,IGH,INC
S1:     A(I) = B(I) + C(I)
S2:     D(I) = A(I-INC)
    end do
```

the same relation $S_1\ \delta_<\ S_2$ holds, but is more difficult to detect, since the values of LOW, IGH, and INC are all unknown to the compiler, and even the sign of INC is unknown (a negative increment would make the loop go backwards). The following is another case in which symbolic data dependence (so called because the subscript expression cannot be decomposed into compile-time constants) is needed:

```
    do I = 1,N
S1:     A(LOW+I-1) = B(I)
S2:     B(I+N) = A(LOW+I)
    end do
```

Here, the two references to the array A can be compared by canceling out the loop-invariant value LOW. This is then the same as comparing A(I-1) to A(I), which can be handled by simpler tests. The two B array references cause no data dependence, since the section of the array referenced by B(I) is B(1:N), which does not intersect with the section of the array referenced by B(I+N), namely B(N+1:N+N).

Global Forward Substitution

Global forward substitution is a transformation that substitutes the right-hand side of an assignment statement for occurrences of the left-hand-side variable, which is especially useful in conjunction with symbolic data dependence. In programs, temporary variables are frequently used to hold commonly used subexpressions or offsets; these variables appear later in the program in array subscripts. Without some kind of global knowledge, the data-dependence tests must assume that the set of subscript values might intersect. For example, the program

```
    NP1 = N+1
    NP2 = N+2
          .
          .
          .
    do I = 1,N
S1:     B(I) = A(NP1) + C(I)
S2:     A(I) = A(I) - 1.
        do J = 2,N
S3:         D(J,NP1)
                = D(J-1,NP2)*C(J) + 1.
        end do
    end do
```

defines two variables, NP1 and NP2 in terms of N. A loop, later in the program, uses NP1 and NP2 in array subscripts. If the compiler does not keep any information about NP1, then it must assume that the assignment of A(I) might reassign A(NP1), and thus there is dependence between S_1 and S_2. However, NP1 was defined to be N+1, and the assignment to A(I) will not ever reach A(N+1), so this is a false dependence. Similarly, if the compiler does not keep information about NP1 and NP2, it must assume that S_3 forms a recurrence. In fact, since NP1 can not equal NP2, the two references to the D array in S_3 are independent, and the J loop can be vectorized or concurrentized. Many compilers perform constant propagation [26, 45, 51], which is a special case of global forward substitution.

Semantic Analysis

Semantic analysis of the program can also help remove data-dependence relations. For instance, in the loop

```
    do I = LOW,IGH
S1:     A(I) = B(I) + A(I+M)
    end do
```

S_1 can be vectorized if $M \geq 0$, but not if $M < 0$. By looking at the surrounding code, the compiler might find an if statement:

```
    if ( M > 0 ) then
       do I = LOW,IGH
S1:       A(I) = B(I) + A(I+M)
       end do
    end if
```

Taking the if statement into account, this loop will not be executed unless $M > 0$; therefore, it can be vectorized. Note that since the iterations are not independent, concurrent code for this loop may still not be appropriate.

Sometimes, even if the `if` statement is not present, the program can be modified to achieve a result similar to the previous transformation. The original loop could be transformed into the following:

```
if ( M>=0 ) then
   do I = LOW,IGH
      A(I) = B(I) + A(I+M)
   end do
else
   do I = LOW,IGH
      A(I) = B(I) + A(I+M)
   end do
end if
```

The `then` part of the `if` statement can now be vectorized. This transformation is called *two-version loops* for obvious reasons.

An alternative to semantic analysis is for the compiler to request information from the user. If the user inputs $M \geq 0$ as an assertion that is true immediately before beginning of execution of the original loop, then the compiler can proceed to vectorize the loop. (Assertions are discussed in greater detail in the section on interaction with the programmer, p. 1198.)

Semantic analysis can also be used to generate vector code for the following loop:

```
      do I = LOW,IGH
S1:      A(I) = A(M)
      end do
```

Since the relative values of `LOW`, `IGH`, and `M` are not known, the data dependence $S_1\ \delta\ S_1$ relation must be assumed. This loop can, however, be executed with a vector instruction. The value `A(M)` is loop invariant, even if `M` falls between `LOW` and `IGH`. The value `A(M)` will at worst be copied to itself; it will not change. The vector code for this statement will give the same results as the serial loop, but the entire statement must be examined to prove this.

Interprocedural Dependence Analysis

When a procedure or function call appears in a loop, most compilers will assume that the loop must be executed serially. Analysis of the effects of the procedure or function call, including which parameters are changed and what global variables are used or changed, can allow dependence testing to decide whether or not the procedure call prevents parallel code from being generated. Studying other information about the parameters, such as values of constants, across procedure call boundaries can help the compiler optimize the code [9, 12, 14, 49].

An alternative method for handling procedure calls is to expand the procedure in-line (also called *procedure integration* [35]. This makes possible the application of some transformations that simultaneously manipulate code in the calling and in the called routines.[9] Also, dependence analysis for the subroutine body is more exact, since only the effects of the one call must be taken into account, and the overhead of the subroutine call is eliminated. In-line expansion is useful even for serial computers.[10] However, in-line expansion should be done with care to avoid an undue increase in the time required for compilation.

Removal of Output and Antidependences

Output dependences and antidependences are, in some sense, false dependences. They arise not because data are being passed from one statement to another, but because the same memory location is used in more than one place. Often, these false dependences can be removed by changing variable names or copying data.

Variable Renaming. Renaming introduces new variable names to replace some of the occurrences of the old variables throughout the program.

In the following program segment

```
S1: A = B + C
S2: D = A + 1
S3: A = D + E
S4: F = A - 1
```

variable `A` is assigned twice. This produces the output dependence $S_1\ \delta^{\circ}\ S_3$ and the antidependence $S_2\ \overline{\delta}\ S_3$. Both of these dependences disappear if a new variable replaces `A` in S_1 and S_2:

```
S1: A2 = B + C
S2: D = A2 + 1
S3: A = D + E
S4: F = A - 1
```

Renaming arrays is a difficult problem in general. Current compilers rename arrays only in very limited cases, if at all.

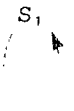
Node Splitting. Some loops contain data-dependence cycles that can be easily eliminated by copying data. The following loop

```
      do I = 1,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = (A(I) + A(I+1)) / 2.
      end do
```

[9] For example, a loop surrounding the subroutine invocation and a loop in the body of the routine could be interchanged (see the section on Loop Interchanging).

[10] Examples of compilers that do in-line substitution are the Experimental Compiling System [4], Parafrase [24], and the Perkin-Elmer Fortran Compiler.

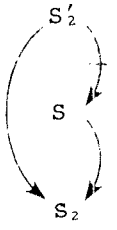
has the following data-dependence graph:



This appears to be a data-dependence cycle. However, one of the arcs in the cycle corresponds to an antidependence; if this arc were removed, the cycle would be broken. The antidependence relation can be removed from the cycle by inserting a new assignment to a compiler temporary array as follows:

```
    do I = 1,N
S2': ATEMP(I) = A(I+1)
S1:  A(I) = B(I) + C(I)
S2:  D(I) = (A(I) + ATEMP(I)) / 2.
    end do
```

The modified loop has the following data-dependence graph:



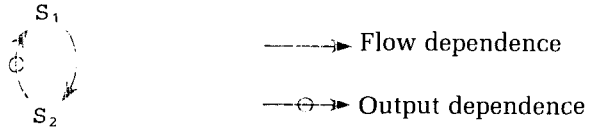
The data-dependence cycle has been eliminated by "splitting" the S_2 node in the data-dependence graph into two parts; the new loop can now be vectorized:

```
S2': ATEMP(1:N) = A(2:N+1)
S1:  A(1:N) = B(1:N) + C(1:N)
S2:  D(1:N) = (A(1:N) + ATEMP(1:N)) / 2.
```

A similar technique can be used to remove output dependences in data-dependence cycles. The loop

```
    do I = 1,N
S1:  A(I) = B(I) + C(I)
S2:  A(I+1) = A(I) + 2*D(I)
    end do
```

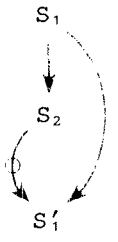
has the following data-dependence graph:



The data dependence cycle can be broken by adding a temporary array:

```
    do I = 1,N
S1:  ATEMP(I) = B(I) + C(I)
S2:  A(I+1) = ATEMP(I) + 2*D(I)
S1': A(I) = ATEMP(I)
    end do
```

The data-dependence graph for the modified loop is



which has no cycles, and can now be vectorized:

```
S1: ATEMP (1:N) = B(1:N) + C(1:N)
S2: A(2:N+1) = ATEMP(1:N) + 2*D(1:N)
S1': A(1:N) = ATEMP(1:N)
```

In both of these cases the added cost is a copy either to or from a compiler temporary array. For machines with vector registers, however, the temporary array will be assigned to a vector register. The "extra" copy is just a vector register load or store that needs to be done anyway; proper placement of the load or store will allow vectorization of these loops without additional statements.

OPTIMIZATIONS FOR VECTOR OR CONCURRENT COMPUTERS

Many of the optimizations in this section were designed with vector or concurrent computers in mind. These optimizations are used to exploit more parallelism or to use parallelism more efficiently on the target computer.

Due to space limitations, the list of methods we discuss is incomplete. Among the absentees are the methods that deal with `while` loops and with recursion. Only recently in the context of developing optimizing compiler methods for Lisp has recursion been carefully considered [22, 23]. `While` loops, on the other hand, are hard to manipulate, and the known transformation techniques are successful only in limited cases.

Scalar Expansion

Vectorizing compilers will promote (or expand) scalars that are assigned in loops into temporary

arrays. This is clearly necessary in order to generate vector code. For instance, the loop

```
    do I=1,N
S1:    X = A(I) + B(I)
S2:    C(I) = X ** 2
    end do
```

can be vectorized by first expanding X into a temporary array, XTEMP

```
    allocate (XTEMP(1:N))
    do I=1,N
S1:    XTEMP(I) = A(I) + B(I)
S2:    C(I) = XTEMP(I) ** 2
    end do
    X = XTEMP(N)
    free (XTEMP)
```

and then generating vector code:

```
    allocate (XTEMP (1:N))
S1: XTEMP (1:N) = A(1:N) + B(1:N)
S2: C(1:N) = XTEMP(1:N) ** 2
    X = XTEMP(N)
    free (XTEMP)
```

The explicit `allocate` and `free` are not necessary on many computers, such as those with vector registers, since the temporary array will exist only in the registers.

When the target machine is a multiprocessor, there is another alternative. Multiprocessor languages (such as Cedar Fortran[11] and Blaze [37]) allow the declaration of iteration-local variables. Thus, the loop

```
doall I=1,N
   real X
   X = A(I) + B(I)
   C(I) = X ** 2
end doall
```

is another valid transformation of the previous loop, as long as X is not used outside the loop in the original program. The `real X` declaration in the `doall` loop means that there will be a separate copy of X for each iteration of the loop. Declaring a scalar variable as iteration local has the same effect as transforming the scalar into an array.

Loop Interchanging

In a multiply-nested loop, the order of the loops may often be interchanged without affecting the outcome [6, 52]. For instance, the loop

```
do J=1,N
   do I=2,N
      A(I,J) = A(I-1,J) + B(I)          (L1)
   end do
end do
```

is equivalent to the following loop:

```
do I=2,N
   do J=1,N
      A(I,J) = A(I-1,J) + B(I)          (L2)
   end do
end do
```

Loop interchanging can translate either of the previous two loops to the other. Interchanging loops is not always possible. The following loop is an example of a loop that cannot be interchanged:

```
do K=2,N
   do L=1,N-5
      A(K,L) = A(K-1,L+5) + B(K)
   end do
end do
```

Figure 1 (p. 1197) illustrates how data-dependence graphs are used to determine when loop interchanging is valid.

Loop interchanging may be used to aid in loop vectorization. For instance, L1 (above) computes a linear recurrence in its inner loop; interchanging it to create L2 allows vectorization of the J index:

```
do I=2,N
   A(I,1:N) = A(I-1,1:N) + B(I)
end do
```

Loop interchanging may also be used to put a concurrent loop on the outside, leading to a better program after loop concurrentization. Thus, loop L2 could be interchanged into L1, which would then be concurrentized to become the following:

```
doall J=1,N
   do I=2,N
      A(I,J) = A(I-1,J) + B(I)
   end do
end doall
```

Fission by Name

Techniques have been developed to handle virtual memory systems, cache memories, and register allocation. The *fission-by-name* transformation tries to break a single DO loop into several adjacent loops. Two statements in the original loop will be in the same resulting loop if there is at least one variable or array referenced by both statements. Fission by name (originally called "distribution of name parti-

[11] Cedar Fortran is the Fortran dialect being designed for the Cedar multiprocessor project at the University of Illinois [21, 31].

tions" [1]) is used to enhance memory hierarchy performance.

Loop Fusion

Loop fusion is a conventional compiler optimization [3, 35] that transforms two adjacent loops into a single loop. The use of data-dependence tests allows fusion of more loops than is possible with standard techniques. For example, the loops

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
      end do
      do I = 2,N
S2:      D(I) = A(I-1)
      end do
```

would not be fused by conventional compilers that do not study the array subscripts. However, the loop fusion is legal since the data-dependence relation $S_1 \ \delta \ S_2$ would not be violated:

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I-1)
      end do
```

A slightly modified example shows when loop fusion is not legal:

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
      end do
      do I = 2,N
S2:      D(I) = A(I+1)
      end do
```

In the original two loops, the data-dependence relation $S_1 \ \delta \ S_2$ holds; the fused loop below, however, has the relation $S_2 \ \overline{\delta} \ S_1$:

```
      do I = 2,N
S1:      A(I) = B(I) + C(I)
S2:      D(I) = A(I+1)
      end do
```

Loop fusion is used in conventional compilers to reduce the overhead of loops. Likewise, fusion helps to reduce start-up costs for `doall` loops. It may also increase overlapping if two `doall` loops require synchronization between iterations.

Since loop fusion and loop fission are dual transformations, any compiler that uses both of them should do so carefully. Loop fusion should not be used to fuse the loops just created by fission. Loop fusion can be used to combine separate loops, if they all refer to the same set of variables, with the same goal as fission by name. For instance, by fusing several loops that refer only to the arrays `{A,B,D}`, the compiler would have a larger loop with the benefits of loop fusion, but still have the improved memory hierarchy performance that comes from only referring to a small set of arrays in the loop.

Strip Mining

Strip mining [35] is used for memory management; it transforms a singly nested loop into a doubly nested one. The outer loop steps through the index set in blocks of some size, and the inner loop steps through each block. As an example, consider the following loop:

```
do I=1,N
   A(I) = B(I) + 1
   D(I) = B(I) - 1
end do
```

After strip mining, this becomes the following:

```
do J=1,N,32
   do I=J,MIN(J+31,N)
      A(I) = B(I) + 1
      D(I) = B(I) - 1
   end do
end do
```

Thus, the loop is excavated in chunks, just as a strip mine is excavated in shovelfuls.

The block size of the outer block loop (32 in this example) is determined by some characteristic of the target machine, such as the vector register length or the cache memory size (see Figure 2, p. 1198, for an example of strip mining and fission by name). For vector machines, the inner strip loop will be vectorized; for parallel computers, the outer block loop can sometimes be concurrentized. Figure 3 (p. 1199) shows how strip mining and loop interchanging can be combined to optimize performance.

Loop Collapsing

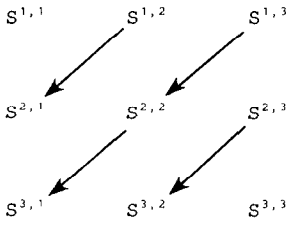
Loop collapsing [44, 52] transforms two nested loops into a single loop, which is used to increase the effective vector length for vector machines. Therefore,

```
real A(5,5),B(5,5)
do I = 1,5
   do J = 1,5
      A(I,J) = B(I,J) + 2.
   end do
end do
```

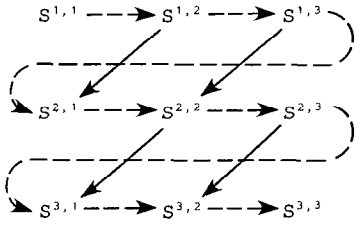
becomes as follows:

```
      do I=1,3
         do J=1,3
S:          A(I,J) = A(I-1,J+1)
         end do
      end do
```

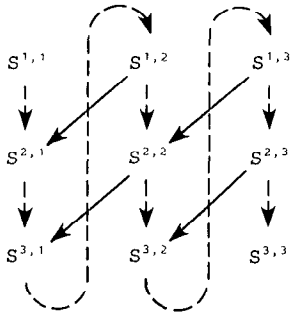
(a)



(b)



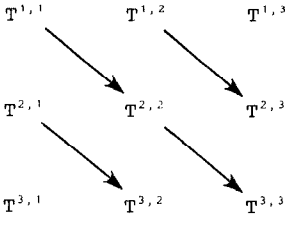
(c)



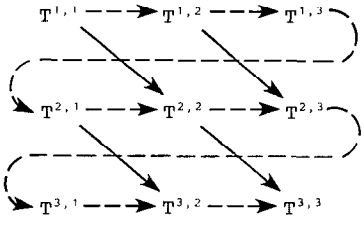
(d)

```
      do I=1,3
         do J=1,3
T:          A(I,J) = A(I-1,J-1)
         end do
      end do
```

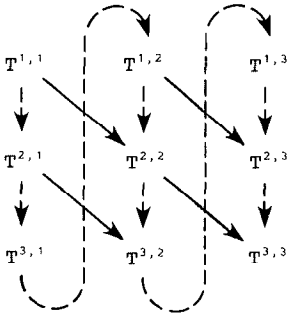
(e)



(f)



(g)



(h)

The compiler uses data dependences to determine when loop interchanging is valid. For example, the loops in (a) may not be interchanged. To see why, consider its iteration space in (b). The arrows in (b) show the data-dependence relations flowing across the iteration space. The instances of statement S are executed by (a) in the order shown by the dotted arrows in (c). If the loops in (a) were interchanged, the new order of execution would be as shown in (d). In this loop ordering, $S^{2,1}$ would be executed before $S^{1,2}$, even though the $S^{2,1}$ needs a value computed by $S^{1,2}$; thus this statement ordering is invalid. As a second example, consider the loops in (e). In this case interchanging is clearly valid since no dependences are violated in the new execution order.

FIGURE 1. How to Determine When Loop Interchanging Is Valid

```
real A(25),B(25)
do IJ = 1,25
   A(IJ) = B(IJ) + 2.
end do
```

A general version of loop collapsing is useful for parallel computers where only a single `doall` nest is supported or to improve the performance of self-scheduled loops. In general, this may require the introduction of some extra assignment statements. For instance, the loop

```
do I=1,N
   do J=1,M
      A(I,J) = B(I,J) + 2.
   end do
end do
```

```
do I=1,N
   A(I) = B(I) + C(I)
   E(I) = F(I) + G(I)
   D(I) = A(I) + B(I)
end do
```

(a)

```
do I=1,N
   A(I) = B(I) + C(I)
   D(I) = A(I) + B(I)
end do
do I=1,N
   E(I) = F(I) + G(I)
end do
```

(b)

```
do J=1,N,32
   do I=J,min(N,J+31)
      A(I) = B(I) + C(I)
      D(I) = A(I) + B(I)
   end do
end do
do J=1,N,32
   do I=J,min(N,J+31)
      E(I) = F(I) + G(I)
   end do
end do
```

(c)

```
do J=1,N,32
   K = min(N,J+31)
   A(J:K) = B(J:K) + C(J:K)
   D(J:K) = A(J:K) + B(J:K)
end do
do J=1,N,32
   K = min(N,J+31)
   E(J:K) = F(J:K) + G(J:K)
end do
```

(d)

```
do J=1,N,32
   K = min(N,J+31)
   vr1 ← B(J:K)
   vr2 ← C(J:K)
   vr3 ← vr1 + vr2
   A(J:K) ← vr3
   vr2 ← vr3 + vr1
   D(J:K) ← vr2
end do
do J=1,N,32
   K = min(N,J+31)
   vr1 ← F(J:K)
   vr2 ← G(J:K)
   vr3 ← vr1 + vr2
   E(J:K) ← vr3
end do
```

(e)

Fission by name and strip mining are used to improve memory performance. Assume a target computer with 32-element vector registers. Before register allocation is performed, the input loop (a) will go through the following sequence of transformations. First, fission by name is applied (b), then strip mining (c), and finally loop vectorization (d). The target program (d) is now a sequence of 32-element vector operations. Registers can now be allocated as shown in (e) (vr1, vr2, and vr3 are vector registers).

Fission by name is useful in decreasing the number of registers needed in a loop. Had the statement `E(I) = F(I) + G(I)` remained in its original position, either more registers or more memory read/writes would have been needed.

FIGURE 2. An Example of Fission by Name and Strip Mining

may be transformed into

```
do L=1,N*M
   I = ⌈ L/M ⌉
   J = mod(L-1,M) + 1
   A(I,J) = B(I,J) + 2.
end do
```

regardless of the bounds of the array A.

INTERACTION WITH THE PROGRAMMER

Several user interaction strategies have been used by optimizing compilers for parallel computers. The most frequent approach is for the compiler to translate directly into object code and to provide a summary specifying what was vectorized and what was not. When something is not vectorized, the compiler gives a reason, which could be the presence of a data dependence, the need to assume a dependence because a subscript range is not known at compile time, the presence of a call to an unknown routine, and so on. If users are not satisfied with the outcome, they may resubmit the program after rewriting parts of it or after inserting directives or assertions.

For example, consider the following loop:

```
do I=1,N
   A(K(I)) = A(K(I)) + C(I)
end do
```

Most compilers will not vectorize this loop; instead they will notify the user that the compiler was forced to assume a dependence since it did not know the value of vector K at compile time. If programmers know that K is a permutation of a subset of the integers, they may order the compiler, through a directive, to vectorize the loop. This directive usu-

```
do I=1,N
   do J=1,N
      do K=1,N
         C(I,J) = C(I,J)
       + A(I,K) * B(K,J)
      end do
   end do
end do
```

(a)

```
do J=1,N
   do K=1,N
      do I=1,N
         C(I,J) = C(I,J)
       + A(I,K) * B(K,J)
      end do
   end do
end do
```

(b)

```
do J=1,N
   do K=1,N
      do L=1,N,64
         do I=L,min(L+63,N)
            C(I,J) = C(I,J)
          + A(I,K) * B(K,J)
         end do
      end do
   end do
end do
```

(c)

```
do J=1,N
   do K=1,N
      do L=1,N,64
         I = min(L+63,N)
         C(L:I,J) = C(L:I,J)
       + A(L:I,K) * B(K,J)
      end do
   end do
end do
```

(d)

```
do J=1,N
   do K=1,N
      do L=1,N,64
         I = min(L+63,N)
         vr1 ← C(L:I,J)
         vr2 ← A(L:I,K)
         sr0 ← B(K,J)
         vr3 ← vr2 * sr0
         vr1 ← vr1 + vr3
         C(L:I,J) ← vr1
      end do
   end do
end do
```

(e)

```
do L=1,N,64
   I = min(L+63,N)
   do J=1,N
      do K=1,N
         vr1 ← C(L:I,J)
         vr2 ← A(L:I,K)
         sr0 ← B(K,J)
         vr3 ← vr2 * vr0
         vr1 ← vr1 + vr3
         C(L:I,J) ← vr1
      end do
   end do
end do
```

(f)

```
do L=1,N,64
   I = min(L+63,N)
   do J=1,N
      vr1 ← C(L:I,J)
      do K=1,N
         vr2 ← A(L:I,K)
         sr0 ← B(K,J)
         vr3 ← vr2 * sr0
         vr1 ← vr1 + vr3
      end do
      C(L:I,J) ← vr1
   end do
end do
```

(g)

The translation of (a) illustrates two interesting uses of loop interchanging. Loop (a) could be vectorized in the form in which it is presented, but it would require a SUM, which may cause round-off error problems. The loops can be interchanged so that I becomes the innermost index (b). After strip mining (c), the innermost loop is vectorized (d). Notice that thanks to loop interchanging the elements of the vectors in (d) are in contiguous memory locations. Vector register assignment may now be performed, leading to (e). The block loop L may be interchanged to become the outermost loop (f). We can now illustrate a second consequence of loop interchanging. The vector register load `vr1 ← C(L:I,J)` and store `C(L:I,J) ← vr1` in (e) are loop invariant and may be moved outside the innermost loop (g). Loop interchanging to increase the number of loop invariant register loads and stores is also a useful sequential optimization technique.

FIGURE 3. Two Examples of Loop Interchanging

ally takes the form of a comment line preceding the loop.

Another way for the user to supply information to the compiler is through assertions. This is sometimes provided as an alternative to compiler directives. For example, in the previous loop the programmer could have asserted that K is a permutation of a subset of the integers. Assertions have two advantages over directives: They are self-explanatory, and they can be tested at run time while debugging the program. Directives, on the other hand, may be a simpler way or even the only way to specify what the user wants. For example, directives may be the only way for the user to request that some part of the code be executed sequentially.

A second user interaction strategy is to produce a restructured source program with vector and/or concurrent language extensions.[12] This strategy makes it possible for the programmer to learn how a program was translated without having to look at the assembly code. These restructurers, like the compilers discussed above, accept directives and assertions from the user.

Experimental interactive restructurers, such as the Blaze Restructurer at Indiana University and the $\mathbb{R}^n$ programming environment at Rice University, are being developed. Some of these restructurers rely on the user to specify the transformations, and users may specify interactively what scalars to expand, what loops to interchange, which ones to vectorize, and so on. These tools make the process of hand-rewriting programs highly reliable and may become a useful tool for program development.

Commercial interactive vectorizers are already available.[13] These interactive tools allow users to find the most time-consuming parts of their programs and rewrite them. They will also aid users in writing vectorizable code in order to get the best performance.

SUMMARY

When the Cray-1 was first delivered to Los Alamos Scientific Laboratories in 1976, there was a great deal of skepticism about whether compiler technology would ever catch up with hardware. Much research had been done for the Texas Instruments ASC and for the Illiac IV on automatic detection of parallelism, but it did not yet meet the needs of the user community. Since that time, several vector supercomputers and minisupercomputers have been announced, each with its own vectorizing compiler. Some of these compilers are more powerful than others, but all are much better than anything hoped for in 1976.

Now, a new generation of computers with multiple processors is coming. Compilers to detect parallelism suitable for spreading over many processors already exist, and more will follow. Many of the optimizations that were used for vectorization are also useful for multiprocessing. That so many of the ideas can be shared by different architectures will make it easier to write compilers for new machines as time passes.

Acknowledgments. The authors would like to thank U. Banerjee, W. L. Harrison, A. Veidenbaum, and the referees for their many helpful comments about this article.

Further Reading

Optimizing compiler algorithms. See [5], [28], [29], [33], [41], and [52].

Dependence Analysis. See [8]–[10].

[12] This is the approach taken by Parafrase, at the University of Illinois, KAP, from Kuck and Associates, PFC, at Rice University, and VAST from Pacific Sierra.

[13] Notably, Forge from Pacific Sierra for the Cray supercomputers, and an interactive vectorizer facility for the Fujitsu supercomputers [20].

REFERENCES

1. Abu-Sufah, W., Kuck, D.J., and Lawrie, D.H. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput. C-30*, 5 (May 1981), 341–356.
2. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
3. Allen, F.E., and Cocke, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1–30.
4. Allen, F.E., Carter, J.L., Fabri, J., Ferrante, J., Harrison, W.H., Loewner, P.G., and Trevillyan, L.H. The experimental compiling system. *IBM J. Res. Dev. 24*, 6 (Nov. 1980), 695–715.
5. Allen, J.R., and Kennedy, K. PFC: A program to convert Fortran to parallel form. Rep. MASC-TR82-6, Rice Univ., Houston, Tex., Mar. 1982.
6. Allen, J.R., and Kennedy, K. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction* (Montreal, June 17–22). ACM, New York, 1984, pp. 233–246.
7. American National Standards Institute *American National Standard for Information Systems. Programming Language Fortran.S8* (X3.9-198x). *Revision of X3.9-1978. Draft S8, Version 99.* American National Standards Institute, New York, Apr. 1986.
8. Banerjee, U. Speedup of ordinary programs. Ph.D. thesis, Rep. 79-989, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Oct. 1979.
9. Banerjee, U. Direct parallelization of call statements—A review. Rep. 576, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Nov. 1985.
10. Banerjee, U., Chen, S.C., Kuck, D.J., and Towle, R.A. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. Comput. C-28*, 9 (Sept. 1979), 660–670.
11. Brode, B. Precompilation of Fortran programs to facilitate array processing. *Computer 14*, 9 (Sept. 1981), 46–51.
12. Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Not. 21*, 7 (July 1986), 162–175.
13. Burroughs Corp. *Numerical Aerodynamic Simulation Facility Feasibility Study.* Burroughs Corp., Paoli, Pa., Mar. 1979.
14. Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L. Interprocedural constant propagation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Not. 21*, 7 (July 1986), 152–161.
15. Chen, S.C. Large-scale and high-speed multiprocessor system for scientific applications: Cray X-MP Series. In *High-Speed Computation*, NATO ASI Series, vol. F7, J.S. Kowalik, Ed. Springer-Verlag, New York, 1984, pp. 59–67.

16. Cytron, R.G. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986, pp. 836–844.
17. Davies, J., Huson, C., Macke, T., Leasure, B., and Wolfe, M. The KAP/S-1: An advanced source-to-source vectorizer for the S-1 Mark IIa supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986, pp. 833–835.
18. Davies, J., Huson, C., Macke, T., Leasure, B., and Wolfe, M. The KAP/205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986, pp. 827–832.
19. Davies, J.R. Parallel loop constructs for multiprocessors. M.S. thesis, Rep. 81-1070, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, May 1981.
20. Dongarra, J.J., and Hinds, A. Comparison of the Cray X-MP-4, Fujitsu VP-200, and Hitachi S-810/20: An Argonne perspective. Rep. ANL-85-19, Argonne National Laboratory, Argonne, Ill., Oct. 1985.
21. Guzzi, M.D. Cedar Fortran Reference Manual. Rep. 601, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Nov. 1986.
22. Harrison, W.L. Compiling LISP for evaluation on a tightly coupled multiprocessor. Rep. 565, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Mar. 1986.
23. Harrison, W.L., and Padua, D.A. Representing S-expressions for the efficient evaluation of Lisp on parallel processors. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986, pp. 703–710.
24. Huson, C.A. An in-line subroutine expander for parafrase. M.S. thesis, Rep. 82-1118, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Dec. 1982.
25. Kamiya, S., Isobe, F., Takashima, H., and Takiuchi, M. Practical vectorization techniques for the Facom VP. In *Information Processing 83*, R.E.A. Mason Ed. Elsevier North-Holland, New York, 1983, pp. 389–394.
26. Kildall, G.A. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages (POPL)* (Boston, Mass., Oct. 1–3). ACM, New York, 1973, pp. 194–206.
27. Kruskal, C.P., and Weiss, A. Allocating independent subtasks on parallel processors. In *Proceedings of the 1984 International Conference on Parallel Processing*, R.M. Keller, Ed. IEEE Press, New York, Aug. 1984, pp. 236–240.
28. Kuck, D.J. Parallel processing of ordinary programs. In *Advances in Computers*, vol. 15, M. Rubinoff and M.C. Yovits, Eds. Academic Press, New York, 1976, pp. 119–179.
29. Kuck, D.J. A survey of parallel machine organization and programming. *ACM Comput. Surv. 9*, 1 (Mar. 1977), 29–59.
30. Kuck, D.J., and Stokes, R.A. The Burroughs scientific processor (BSP). *Special Issue on Supersystems, IEEE Trans. Comput. C-31*, 5 (May 1982), 363–376.
31. Kuck, D.J., Davidson, E.S., Lawrie, D.H., and Sameh, A.H. Parallel supercomputing today and the Cedar approach. *Science 231*, 4740 (Feb. 28, 1986), 967–974.
32. Kuck, D.J., Kuhn, R.H., Leasure, B., and Wolfe, M. The structure of an advanced retargetable vectorizer. In *Tutorial on Supercomputers: Designs and Applications*, K. Hwang, Ed. IEEE Press, New York, 1984, pp. 163–178.
33. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL)* (Williamsburg, Va., Jan. 26–28). 1981, pp. 207–218.
34. Kuck, D.J., Sameh, A.H., Cytron, R., Veidenbaum, A.V., Polychronopoulos, C.D., Lee, G., McDaniel, T., Leasure, B.R., Beckman, C., Davies, J.R.B., and Kruskal, C.P. The effects of program restructuring, algorithm change, and architecture choice on program performance. In *Proceedings of the 1984 International Conference on Parallel Processing*, R.M. Keller, Ed. IEEE Press, New York, Aug. 1984, pp. 129–138.
35. Loveman, D.B. Program improvement by source-to-source transformation. *J. ACM 24*, 1 (Jan. 1977), 121–145.
36. Lundstrom, S.F., Barnes, G.H. A controllable MIMD architecture. In *Proceedings of the 1980 International Conference on Parallel Processing* (Bellaire, Mich., Aug. 26–29). IEEE Press, New York, 1980, pp. 19–27.
37. Mehrotra, P., and Van Rosendale, J. The Blaze Language: A parallel language for scientific programming. Rep. 85-29, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Va., May 1985.
38. Midkiff, S.P., and Padua, D.A. Compiler generated synchronization for Do loops. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986.
39. Miura, K., and Uchida, K. Facom vector processor VP-100/VP-200. In *High-Speed Computation*, NATO ASI Series, vol. F7, J.S. Kowalik, Ed. Springer-Verlag, New York, 1984, pp. 127–138.
40. Nagashima, S., Inagami, Y., Odaka, T., and Kawabe, S. Design consideration for a high-speed vector processor: The Hitachi S-810. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers. ICCD 84.* (Port Chester, N.Y., Oct. 8–11). IEEE Press, New York, 1984, pp. 238–243.
41. Ottenstein, K.J. A brief survey of implicit parallelism detection. In *MIMD Computation: The HEP Supercomputer and its Applications*, J.S. Kowalik, Ed. MIT Press, Cambridge, Mass., 1985.
42. Padua, D.A. Multiprocessors: Discussion of some theoretical and practical problems. Ph.D. thesis, Rep. 79-990, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Oct. 1979.
43. Padua, D.A., Kuck, D.J., and Lawrie, D.H. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comput. C-29*, 9 (Sept. 1980), 763–776.
44. Polychronopoulos, C.D. Program restructuring, scheduling, and communication for parallel processor systems. Ph.D. thesis, Rep. 595, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1986.
45. Reif, J.H., and Lewis, H.R. Symbolic evaluation and the global value graph. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL)* (Los Angeles, Calif., Jan. 17–19). ACM, New York, 1977, pp. 104–118.
46. Scarborough, R.G., and Kolsky, H.G. A vectorizing Fortran compiler. *IBM J. Res. Dev. 30*, 2 (Mar. 1986), 163–171.
47. Tang, P., and Yew, P. Processor self-scheduling for multiple-nested parallel loops. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). IEEE Press, New York, 1986, pp. 528–535.
48. Thornton, J.E. *Design of a Computer: The Control Data 6600.* Scott, Foresman and Co., Glenview, Ill., 1970.
49. Triolet, R., Irigoin, F., and Feautrier, P. Direct parallelization of call statements. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Not. 21*, 7 (July 1986), 176–185.
50. Veidenbaum, A. Program optimization and architecture design issues for high-speed multiprocessors. Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1985.
51. Wegman, M., and Zadek, K. Constant propagation with conditional branches. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL)* (New Orleans, La., Jan. 14–16). ACM, New York, 1985, pp. 291–299.
52. Wolfe, M.J. Optimizing supercompilers for supercomputers. Ph.D. thesis, Rep. 82-1105, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Oct. 1982.
53. Yasumura, M., Tanaka, Y., Kanada, Y., and Aoyama, A. Compiling algorithms and techniques for the S-810 vector processor. In *Proceedings of the 1984 International Conference on Parallel Processing*, R.M. Keller, Ed. IEEE Press, New York, Aug. 1984, pp. 285–290.

CR Categories and Subject Descriptors: C.1.2 **[Processor Architectures]**: Multiple Data Stream Architectures (Multiprocessors)—*array and vector processors; multiple-instruction-stream, multiple-data-stream processors (MIMD); parallel processors; pipeline processors; single-instruction-stream, multiple-data-stream processors (SIMD)*; D.2.7 **[Software Engineering]**: Distribution and Maintenance—*restructuring*; D.3.3 **[Programming Languages]**: Language Constructs—*concurrent programming structures*; D.3.4 **[Programming Languages]**: Processors—*code generation; compilers; optimization; preprocessors*

General Terms: Languages, Performance

Authors' Present Addresses: David A. Padua, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801; Michael J. Wolfe, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; and Kuck and Associates, 1808 Woodfield, Savoy, IL 61874.