A vectorizing Fortran compiler

by Randolph G. Scarborough
Harwood G. Kolsky

This paper describes a vectorizing Fortran compiler for the IBM 3090 Vector Facility. Opportunities for vectorization are investigated for eight levels of DO-loop nesting. Recurrences in inner loops do not prevent vectorization of outer loops. A least-cost analysis determines from the opportunities identified which specific vectorization will result in the fastest execution. The normal optimization phases of the compiler produce much of the information needed for the vectorization analysis.

Introduction

Many programs written for computers contain loops, and nests of loops, for performing repetitive operations on sequences of data. These programs direct that operations be done in a well-defined order. Because scalar machines have historically been the most widely available type of machines, the order is one that is readily executable on a scalar machine. On a vector machine, however, where successive elements in the order are processed in parallel, this very same order may not be valid. There may exist other orders in which the elements may be processed correctly, but analysis is required to discover both the valid orders and the parts of programs for which the vector machine may be used. This analysis and its result is commonly known as vectorization. The purpose is to begin with a program written for a scalar machine and make it execute on a vector machine.

Vectorization is of interest for at least three reasons. First is the large existing body of programming written for scalar machines. An automatic vectorizer makes it possible for an existing program to be executed on a vector machine without requiring that it be inspected or rewritten. Second is portability. An automatic vectorizer makes it possible to execute a program without change on both a scalar and a vector machine. Third is speed. An automatic vectorizer makes it possible for a program to be optimized for scalar when it is run on a scalar machine and to be optimized for vector when it is run on a vector machine.

The objectives of the vectorizing compiler, then, are these:

- To identify as many as possible of the source program statements which may be vectorized.
- To identify which of the loops surrounding these statements may be used to vectorize them.
- To select, from all of the resulting possible choices for vectorization, the one which will result in the fastest execution of the program.
- To compile highly optimized vector object code for the statements chosen to be run in vector hardware.
- To leave the rest of the program undisturbed in highly optimized scalar object code.
- To do the above without requiring changes to the original source program.

In this paper vectorization is discussed throughout in terms of the Fortran language. Fortran is the language of interest. However, the vectorization algorithms and techniques may be applied to other languages as well.

The remaining sections of this paper contain

- A review of program dependence theory, which underlies the vectorization algorithms.
- A review of some work done at Rice University which provided the foundation for this work.
- A description of the structure of the vectorizing compiler.
- A description of the compiler algorithms which determine what statements are vectorizable at what levels.
- A description of the compiler algorithms which determine what statements are chosen for vector execution at what levels.
- Some examples of vectorization.

Program dependence

Vectorization is based on program dependence theory. The base for this vectorizer is the work of Kennedy [1] and Allen et al. [2–5]. Their work in turn builds on earlier work of Kuck et al. [6] and Banerjee [7, 8]. These references should be consulted for a precise discussion of program dependence. In this paper the topic is summarized at an intuitive level.

In general, a statement in a nest of DO-loops may be vectorized if it does not require, as an input on one iteration of a loop, a value it computed on an earlier iteration of the loop. When a value computed in one iteration of a DO-loop is not used in a later iteration, all of the data values can be computed in parallel. This independence of data values from one DO-loop iteration to the next is a key factor in allowing execution of the statement on a vector machine.

The fundamental vectorization algorithm is to determine the dependences between the statements in a program and to construct a graph representing these dependences. Those statements not in strongly connected regions of this graph are vectorizable. Those statements which do lie in strongly connected regions are not vectorizable because they depend, possibly via some sequence of intermediate statements, on values they themselves compute.

• *Data dependence*

The dependence that may arise when two statements reference the same storage location is called data dependence. Data dependences arise in one of three ways:

1. A statement T depends upon a statement S when S defines a value and T references it. This is called true dependence.

```
S:  X=
T:   =X
```

   Clearly, S must execute before T can execute because S defines a value used by T.
2. A statement T depends upon a statement S when S references a value and T defines it. This is called anti-dependence.

```
S:   =X
T:  X=
```

   Again, S must execute before T because otherwise T would store the variable X and S would use the wrong value.
3. A statement T depends upon a statement S when S stores a value which T also stores. This is called output dependence.

```
S:  X=
T:  X=
```

   S must execute before T or else the wrong value will be left behind in the variable X.

• *Control dependence*

A dependence may arise when one statement determines whether a second statement will be executed. This is called control dependence:

```
  DO 3 I=1,N
1 IF (A(I).GT.0.0) GOTO 3
2 A(I)=B(I)+1.0
3 CONTINUE
```

Clearly, statement 1 must execute before statement 2 can execute. Statement 2 depends upon statement 1.

Control dependences may be changed into data dependences with a technique called IF-conversion. The following example is equivalent to the preceding example:

```
  DO 3 I=1,N
1 L=A(I).LE.0.0
2 IF (L) A(I)=B(I)+1.0
3 CONTINUE
```

Statement 2 still depends on statement 1, but now the dependence arises because of a true data dependence involving the variable L. More complex transformations are required to handle more general control structures. (The compiler described here processes only forward branches which do not cross loop boundaries.) The transformations permit the vectorizer to operate by evaluating data dependences only.

• *Dependence level*

Dependences attach to particular DO-loop levels in the loops surrounding a group of statements. Some dependences are always present:

```
    DO J=
    DO I=
S:  V(I,J)=A(I,J)*B(I,J)
T:  Z(I,J)=V(I,J)
```

T always depends upon S because, on every iteration of every loop, there is a true dependence involving the variable V. These dependences are called loop-independent dependences: they are independent of the operation of the loops surrounding them.

Some dependences are present at one loop level but not another:

```
    DO J=
    DO I=
S:  A(I+1,J)=A(I,J)
```

There is a true dependence at the level of the loop with index I; an element stored on iteration 2, for example, will be fetched on iteration 3. But there is no dependence at level J, since no element stored on one iteration of the loop is referenced on any other iteration.

• *Dependence interchange*

When a given loop in a nest of DO-loops is chosen for execution in vector hardware, each vector instruction will operate on successive data elements selected by that given DO-loop index. For example, if the J-loop (the loop with index J) were vectorized in the nest

```
  DO 1 K=1,N
  DO 1 J=1,N
  DO 1 I=1,N
1 A(I,J,K)=A(I+1,J+2,K+3)
```

then vector instructions would fetch the elements of A in the order (2,3,4), (2,4,4), ···, (2,N+2,4) and store them in the order (1,1,1), (1,2,1), ···, (1,N,1). This is a different order from that which would be used in scalar mode, where the innermost DO-loop, with index I, would cycle most rapidly.

In fact, the vector order is exactly what would be seen in scalar mode if the J-loop were interchanged with its inner neighbors until it became the innermost loop:

```
  DO 1 K=1,N
  DO 1 I=1,N
  DO 1 J=1,N
1 A(I, J,K)=A(I+1,J+2,K+3)
```

In order for a given loop to be chosen for execution in vector hardware, this interchange must be valid. That is, it must preserve the semantics of the program.

For the K-loop in the original example to be vectorizable, the loop ordering

```
  DO 1 J=1,N
  DO 1 I=1,N
  DO 1 K=1,N
1 A(I,J,K)=A(I+1,J+2,K+3)
```

would have to generate the same program results as the original. Note that the other loops are not permuted. It is necessary to ask only if the loop of interest may be moved inside all of the others.

Sometimes this loop interchange is not possible. In the nest

```
  DO 1 J=1,N
  DO 1 I=1,N
1 A(I-1,J+1)=A(I,J)
```

there is a dependence at the level of the J-loop. A value stored on one iteration of J is fetched on the next. Many dependences do not affect the results of the program when loops are interchanged. But this one does, and the J-loop cannot be interchanged with the I-loop because the answers would change. This is an interchange preventing dependence, and it prevents vectorization of the J-loop.

In a multi-level nest, a dependence for a loop at some level might be interchangeable part of the way into the innermost level, but then be blocked. Such a dependence is called "innermost preventing" because the loop at that level cannot be moved to the innermost level (or be "innermosted"). If the loop cannot be innermosted then it cannot be vectorized. Innermosting will be described in more detail later.

Parallel Fortran converter

The prototype for the present work was a program known as the Parallel Fortran Converter written at Rice University [2]. This program, under continuous improvement over the years 1980–1984, was an evolving project of Professor Kenneth W. Kennedy and his students. The goal of the program was to translate statements written in scalar Fortran 77 into equivalent statements written in a vector Fortran language known generically as Fortran 8x. The dependence theory was developed and elaborated in pursuit of this goal. Among the significant innovations were the characterization of dependence by DO-loop nesting level and the development of techniques for IF-conversion.

Our compiler work was strongly directed by a version of the Converter current in 1982. Many parts of the compiler still show direct connections to this valuable work at Rice. For this reason a summary of the Parallel Fortran Converter is useful.

The Converter vectorized programs through a three-stage process. In the first stage, called subscript standardization, the program was placed into a standard form for dependence testing. DO-loops were normalized to run from 1 in increments of 1 to some upper limit. User-written induction variables were replaced by equivalent references to the DO-loop variables. Subscripts were simplified.

The second stage was dependence testing. Every pair of statements which might have a dependence was examined by several algorithms designed to prove, if possible, that the statements were independent. The most important of these tests was the Banerjee test [7] as elaborated by Kennedy [1]. The tests all required as input the normalized DO-loop parameters surrounding the statements and the normalized subscript components for the variables which might give rise to a dependence. The tests all returned as output a decision for each DO-loop level that the statements were provably independent or not at that level. When the statements could not be proved to be independent then they were assumed to be dependent. As the dependence testing was performed a directed graph was constructed. Each statement was a node in the graph. Each dependence became an edge, marked with the level of the dependence, from the source to the target of the dependence.

The third and final stage was parallel code generation. The dependence graph was partitioned into strongly connected regions. There is by definition a path (but not necessarily a direct path) from every node to every other node in a strongly connected region. Since the nodes represented

statements and the path was formed from edges which represented dependences, this implied that every statement in the region depended directly or indirectly upon itself. Such a dependence of a statement upon itself is known as a recurrence. Those statements which were not in strongly connected regions and therefore not in recurrences could be vectorized, and parallel Fortran statements were generated for them. Those statements which were in strongly connected regions could not be vectorized since they depended upon values that they themselves computed. Instead, a scalar DO-statement was generated for the outermost DO-loop represented in the graph, the edges in the graph representing dependences at this level were deleted, and the statements in the region were then recursively partitioned at inner levels in this same manner.

One refinement occurred at the next-to-innermost level. When statements were strongly connected at this level, the Converter interchanged the inner two loops to determine what statements were strongly connected with the revised loop order. It then picked for its final loop order the one of these two cases which produced more vectorized statements.

For our purposes, however, this process posed some problems. First, scalar DO-loops were placed around statements containing recurrences regardless of the level at which the dependences causing the recurrences arose. A recurrence at an inner level caused all outer levels to become not vectorizable. Second, decisions about the final state of the code were made incrementally as the analysis moved from outer to inner levels. No exploration of alternatives was performed (except for the innermost two levels) before a level was committed to scalar or vector code. There was no provision for evaluating alternatives, when alternatives existed, to decide which was best. Finally, these decisions and the program normalization which preceded them were performed by irrevocably transforming the program being analyzed. IF-conversion in particular could introduce staggering overhead into code which failed to vectorize.

Structure of the vectorizing compiler

Our objective was to imbed the vectorizer analysis into a production compiler rather than a source-to-source converter. This implied that the vectorizer had to be fast. If it were very slow, then a separate source-to-source converter might make sense, since in that case the cost of vectorization would be paid only once. But then a user would be left with two programs, one scalar and one vector, and the benefits of portability would be lost.

The fact that speed was critical to success was obvious from the beginning. A very early example by Kennedy, using vectorization code obtained elsewhere, required 10 CPU minutes on an IBM Model 168 to vectorize a 300-statement program. The Parallel Fortran Converter required 20 seconds for this program [2]. A simple compile, however, took less than a second. Thus it seemed likely that vectorization might increase compilation times by a factor of 25. To achieve the objective of integrating vectorization and compilation would therefore require dramatic increases in performance.

The compiler chosen to contain the vectorization was VS Fortran (although a prototype was built on the base of the Fortran H Extended Optimization Enhancement). The VS Fortran compiler, which is IBM's System/370 Fortran compiler, supports both Fortran 66 and Fortran 77 and produces highly optimized scalar object code. For a detailed description of the optimizations see our earlier paper [9].

Three things were done to integrate the vectorizer into this compiler and to achieve a great increase in speed.

First, vectorization was put into the optimizer, making it a new phase between text and register optimization. This seemed intuitively to be the natural position for the vectorizer, and it has worked well in practice. One reason is that the vectorizer operates on much smaller quantities of code after optimization. Many expressions have been eliminated by common expression elimination; many have been moved outside of loops by backward movement. What remains is usually the unique, necessary, nearly minimum computation.

A more important reason is that the process of optimization in the compiler nearly duplicates that of normalizing the subscripts in preparation for vectorization. The Parallel Fortran Converter, as mentioned above, had a first phase whose task was standardizing subscript information. While the identities between its operations and the compiler optimizations were not exact, the optimizer could be modified to make them very similar. Most of the overhead of preparing to analyze the program was eliminated in this way.

A final reason for placing vectorization after optimization was that vectorization would then not affect the optimization of programs or portions of programs found to be not vectorizable.

The second thing done while integrating the vectorizer into the compiler was to reengineer the algorithms in the Parallel Fortran Converter. These algorithms, which were the result of original research, had been written to make them clear, easily modifiable, and programmable by different people working at different times. One improvement, therefore, was simply to rewrite them with a new focus on speed. Another was to replace inefficient algorithms with much more efficient versions. We call this work reengineering because the algorithms, old and new, compute the same results from the same input. Since we were seeing the algorithms with fresher eyes, and since we were trying not to discover what was needed but to find an efficient way to compute something which was already known to be needed, shortcuts became apparent.

The third major thing done during integration of the vectorizer and the compiler was to change the

implementation language from PL/I (used in the Converter) to Fortran. (The language has subsequently been changed to PL/S. The Fortran version is perhaps 20–25% faster than the PL/S version while running in the converted modules. Both are much faster than the PL/I version.)

As a result of this work, vectorization is now fast enough to be incorporated into a production compiler. **Table 1** shows compile times for various programs with three sets of options: no optimization, maximum optimization, and maximum optimization and vectorization. The programs are a collection of scientific applications collected over time; some vectorize well and others do not. These measurements indicate that vectorization is costing on the average a 23% increase in the compile time for optimized programs, less than the time required by optimization itself.

Table 1 Normalized compile times for various programs.

	Opt(0)	*Opt(3)*	*Vector*
AIRFLOW	0.39	1.00	1.35
AIRMESH	0.58	1.00	1.15
DAMTESTA	0.48	1.00	1.29
DAMTESTB	0.62	1.00	1.09
DEBDASDI	0.94	1.00	1.05
DEBEEEEE	0.71	1.00	1.06
DEBEIGEN	0.35	1.00	1.49
DEBEIGER	0.42	1.00	1.63
DEBFFTXX	0.64	1.00	1.25
DEBGRAPH	0.55	1.00	1.10
DEBHORNE	0.70	1.00	1.05
DEBLLSQE	0.54	1.00	1.35
DEBLSQEE	0.55	1.00	1.29
DEBMINIV	0.55	1.00	1.27
DEBODEER	0.56	1.00	1.20
DEBPFITS	0.61	1.00	1.34
DEBPIEEE	0.71	1.00	1.06
DEBSORTS	0.69	1.00	1.04
GASCOAL	0.58	1.00	1.09
GAZEXEC	0.48	1.00	1.73
GAZINIT	0.60	1.00	1.06
GAZTEST	0.52	1.00	1.32
GFDMARK	0.72	1.00	1.18
GSITESTA	0.56	1.00	1.17
GSITESTB	0.34	1.00	1.17
HHWTEST	0.44	1.00	1.22
KGNVATD	0.35	1.00	1.42
LFPTEST	0.77	1.00	1.05
NASAVA3D	0.40	1.00	1.67
PFHTEST	0.73	1.00	1.05
PPLTEST	0.48	1.00	1.43
RGETEST	0.58	1.00	1.08
SAWTEST	0.75	1.00	1.15
TAHTEST	0.52	1.00	1.11
Average	0.57	—	1.23

Vectorization analysis—Eligibility for vectorization

The goal of the vectorizing compiler is to compile vector object programs from scalar source programs. This is a different goal from that of the Parallel Fortran Converter, which was to compile vector source from scalar source. This latter goal leads into at least two blind alleys. First, some loops, such as

```
  DO 1 I=1,N
1 A(I)=B(J(I),K(I))
```

are not expressible in vector language even though they are recognizable as vectorizable by the dependence analysis and are readily executable on a vector machine. The Converter is forced to generate scalar source code for such loops. Second, the dependence analysis intended for source-to-source conversion does not recognize many cases where vector hardware may be used but, because of an interaction between the semantics of the proposed vector language assignment statement and the dependence algorithm, vector source language may not be used. For example, the loop

```
  DO 1 K=1,32
  DO 1 J=1,32
  DO 1 I=1,32
  A(K+1,J+1,I+1)=B(K+1,J,I+1)+B(K+1,J+1,I)
1 B(K+1,J+1,I+1)=A(K+1,J,I+1)+A(K+1,J+1,I)+A(K,J,I)
```

contains a recurrence at level J and at level I and so the Converter, finding a recurrence when it looks at level K, at level J, and at level I, is forced to leave the loop in scalar code. But there is no reason why the vector hardware cannot be used to execute level K. The dependence algorithm, however, could not generate a vector assignment for level K, and so opportunities for vectorization were being obscured.

Recall for a moment the basic algorithm of the Parallel Fortran Converter. After standardizing the program, it forms a dependence graph, it partitions the graph into strongly connected regions, and then it generates either, for those statements not in strongly connected regions, vector code for all remaining DO-loop levels or, for those statements in strongly connected regions, a scalar DO-loop at the level defining the region, after which it recursively executes this same procedure for the remaining inner DO-loop levels. In other words, whenever the region contains a recurrence, a scalar DO statement, which is not executable in vector, is always generated for the level defining the region, and attention is shifted to opportunities at any remaining inner levels. The problem is that any inner recurrence affects all outer levels, whether or not those outer levels carry a dependence which contributes to the recurrence.

It is with the discovery of what we have called innermosting and its implications for vectorization that our contribution to vectorization began. Our dependence algorithm forms a dependence graph essentially equivalent to the one in the Converter, and it likewise partitions the program into strongly connected regions, level by level, using this graph. But for each of these strongly connected regions it asks one additional question. In spite of the fact that the region is strongly connected, and therefore contains a

recurrence, is it possible nonetheless to use the vector hardware to execute the DO-loop at the level which defines the region?

The algorithm to determine whether a strongly connected region may nonetheless be executed in vector hardware for the DO-loop defining the region has two phases. In the first phase, the algorithm examines each dependence carried by the DO-loop defining the region. It attempts to interchange this DO-loop with each successive inner DO-loop until it has become the innermost loop. If an interchange is not possible, because it would violate the dependence, then the region may not be executed in vector mode at the level defining the region and the method terminates for this region for this level. If interchange is possible, but the dependence itself disappears because it is absorbed by a dependence at an intervening loop, then the dependence is ignored since it will not affect the vector execution of the level defining the region. Otherwise the dependence is recorded as an "innermost sensitive" dependence for the region.

The earlier example may be modified to illustrate these dependences:

```
      DO 1 K=1,32
      DO 1 J=1,32
      DO 1 I=1,32
        A(K+1,J+1,I+1)=B(K+1,J,I+1)+B(K+1,J+1,I)+B(K,J,I)
1     B(K+1,J+1,I+1)=A(K+1,J,I+1)+A(K+1,J+1,I)+A(K,J,I)
```

There are true dependences at level K from A(K+1,J+1,I+1) to A(K,J,I) and from B(K+1,J+1,I+1) to B(K,J,I). When, however, the DO-loop ordering is changed from K,J,I to J,I,K in order to test whether level K may be innermosted, these dependences disappear. In the new ordering they are dependences at level J, not level K, and therefore they are not innermost sensitive dependences for level K. In contrast, the true dependences at level J from A(K+1,J+1,I+1) to A(K+1,J,I+1) and from B(K+1,J+1,I+1) to B(K+1,J,I+1) remain at level J as level J is innermosted, and they are therefore innermost sensitive dependences for level J.

In the second phase, the algorithm constructs a graph using only the loop-independent dependences in the region and the innermost sensitive dependences for the DO-loop level defining the region. Other dependences, notably those for other DO-loop levels, are excluded. If this graph has a cycle then the region may not be executed in vector mode at the level defining the region. Otherwise the region is eligible for vector execution even though other loops inside the loop defining the region may have recurrences. Because many edges have been excluded from the graph, it becomes much more likely that the graph will not be strongly connected and therefore that the region will be found eligible for execution on the vector hardware. The algorithm marks the region if it may be executed in vector mode and then it recursively considers inner regions, marking thereby each region found to be vectorizable.

The foregoing ignores some special cases of recurrences which can be vectorized. These special cases are handled by both vectorizers. For example, reduction operations, such as S=S+A(I,J,K), can be vectorized if reduction hardware exists on a machine. Likewise, techniques such as scalar expansion are used by both vectorizers to reduce or eliminate some recurrences.

Vectorization analysis—Economics of vectorization

In all of the foregoing analysis the vectorizer has made no irrevocable decisions as to the final form of the object program. In particular, it has not written scalar DO-loops, and it has not partitioned code originally from one nest of DO-loops into separate nests, some vector and some scalar. It has done nothing except investigate possibilities, identifying possible choices for use of the vector hardware. Some of these will be the choices identified by the Converter, but others will be new ones, outer loops containing inner loops with recurrences. The vectorizer now seeks to find the fastest possible execution of the program, using vector or scalar hardware as appropriate.

A nest of loops may in general be executed either by scalar instructions or by vector instructions with the vector hardware applied to any of, but only one of, the loops in the nest. For example, in the nest

```
      DO 1 K=1,N
      DO 1 J=1,N
      DO 1 I=1,N
        A(I,J,K)=B(I,J,K)+P(J,K)*Q(J,K)
1     E(K,J,I)=F(K,J,I)+X(I,J)*Y(I,J)
```

four possibilities exist for each statement (vectorize on K, J, I, or none) and sixteen possibilities therefore exist for the combination of the two statements. The vectorizer attempts to find the fastest among these possibilities.

The main factors considered in estimating the cost of execution are the cost of loop overhead, the cost of hardware instructions, and the cost of fetching and storing operands. These costs will vary for the same statement as each enclosing loop is considered as the candidate for execution in the vector hardware. The costs will vary because statements will be scalar at some levels but vector at other levels, and because array operands will be fetched and stored in different dimensions as different levels are considered for vectorization.

The possibilities for scalar and vector execution are treated as a graph. Using a modified least-cost graph traversal algorithm, the vectorizer attempts to find the cheapest path through the graph of possibilities. Since some heuristics are employed the traversal is not exact. Heuristics are necessary because, for example, two statements from the same original nest of loops may have been partitioned into different regions when vector execution candidates were identified

(because they were in different strongly connected regions of the program), but when the least-cost object program is constructed for these statements it is desirable (but not always possible) to merge them back into the same set of loop controls.

The regions identified in the earlier part of the algorithm, when each was marked as eligible or not for vector execution, are sorted into topological order based on program dependence as part of that process. The least-cost graph traversal considers these regions in topological order beginning with a null first region and ending with a null final region. The algorithm produces and maintains a list of the execution costs, in units of processor time, required to execute the statements in subsets of these regions, the subsets always beginning with the null first region in the topological ordering, always ending with some other region in the topological ordering, and always including all regions between the two.

Each element on the list represents the execution of the regions on the path from the null first region through the last region on the path, with each region executed in a particular mode, vector or scalar. The algorithm estimates the cost for executing this entire collection of regions in these particular modes, attempting heuristically to minimize the number of loop control statements which must be inserted. This requires merging the statements on the path into common enclosing loops while preserving the dependences between the statements.

Once the cost has been determined, the next possible candidate regions for execution along this path are identified. These candidates will be the next region in the topological ordering executed in scalar and, if valid for the region, vector. These candidates, and the just-estimated cost of execution to reach them, are then posted onto the list of pending possible execution paths. When a candidate region is marked as a scalar candidate, inner regions of that candidate subsequently are considered as candidates, either scalar or vector. When a candidate region is marked as a vector candidate, by implication all inner regions are scalar and they are bypassed to expedite processing.

The algorithm iterates by picking from all elements on the list the one with the next candidate which is most cheaply reachable along its particular path of execution possibilities, by computing the cost of including it in execution with its predecessors, by locating its possible successors, and by posting them on the list of pending candidates. The algorithm terminates when the null final region is selected as the cheapest candidate to include on a path, since this path represents the cheapest execution which includes all regions in the program.

The selected path represents a decision for each region whether that region is to be executed in scalar or vector mode and indicates how the regions are to be merged into fewer regions for the purposes of minimizing the loop

(a)

```
      DO 1 K=1,32
      DO 1 J=1,32
      DO 1 I=1,32
      AA(I,J,K)=BA(I,J,K)+CA(I,J,K)/DA(I,J,K)
      AB(I,K,J)=BB(I,K,J)+CB(I,K,J)/DB(I,K,J)
      AC(J,I,K)=BC(J,I,K)+CC(J,I,K)/DC(J,I,K)
      AD(J,K,I)=BD(J,K,I)+CD(J,K,I)/DD(J,K,I)
      AE(K,I,J)=BE(K,I,J)+CE(K,I,J)/DE(K,I,J)
    1 AF(K,J,I)=BF(K,J,I)+CF(K,J,I)/DF(K,J,I)
```

(b)

```
VECT 1 ,------ DO 1 K=1,32
SCAL 2 |,----- DO 1 J=1,32
SCAL 3 ||,---- DO 1 I=1,32
     4 |||     AE(K,I,J)=BE(K,I,J)+CE(K,I,J)/DE(K,I,J)
     4 |||     AF(K,J,I)=BF(K,J,I)+CF(K,J,I)/DF(K,J,I)
SCAL 1 |-------DO 1 K=1,32
SCAL 2 |,----- DO 1 J=1,32
VECT 3 ||,---- DO 1 I=1,32
     4 |||     AA(I,J,K)=BA(I,J,K)+CA(I,J,K)/DA(I,J,K)
     4 |||     AB(I,K,J)=BB(I,K,J)+CB(I,K,J)/DB(I,K,J)
VECT 2 |,----- DO 1 J=1,32
SCAL 3 ||,---- DO 1 I=1,32
     4 |||     AC(J,I,K)=BC(J,I,K)+CC(J,I,K)/DC(J,I,K)
     4 |||     AD(J,K,I)=BD(J,K,I)+CD(J,K,I)/DD(J,K,I)
```

Figure 1

Vectorization along different dimensions: (a) original source program; (b) vectorized program.

control statements. The intermediate language for the program is then transcribed to implement these decisions.

Note that scalar code is unaffected by this vectorization algorithm. Optimized scalar code remains optimized scalar code. No changes to the code are made until it has been determined that the changes are not only an improvement but also the best improvement the vectorizer can find.

Examples of vectorization

The compiler reports how it vectorizes a source program by reprinting the original source statements and enclosing them in bracketed loops which show the final object program form. The loops are marked as vector or scalar. Any reordering of statements or replication of loops is indicated in this vector report.

The report is only approximate. Statements which have been optimized away, or moved backward outside loops, may not be seen. Statements manipulating user-written induction variables are especially likely to disappear as a result of optimization.

The vector report is used in the examples to illustrate the vectorization.

Figure 1 shows how statements from the same loop can be vectorized along different dimensions as a result of the economic analysis. Storage is accessed most efficiently, because it is accessed consecutively, when the leftmost subscript of an array is varied most rapidly. This is reflected in the economics. In the original scalar example, which shows the same subscripts permuted six ways, the storage used by four of the six statements is not accessed consecutively. In vector mode, however, the storage for all

(a)

```
      DO 1 K=1,32
      DO 1 J=1,32
      DO 1 I=1,32
      A(I+1,J,K)=1/A(I,J,K)
      B(I+1,J+1,K)=B(I,J+1,K)+B(I+1,J,K)
    1 C(I+1,J+1,K+1)=C(I,J+1,K+1)+C(I+1,J,K+1)+C(I+1,J+1,K)
```

(b)

```
SCAL 1 ------ DO 3 K=1,32
SCAL 2 |----- DO 2 J=1,32
SCAL 3 ||---- DO 1 I=1,32
     4 |||    C(I+1,J+1,K+1)=C(I,J+1,K+1)+C(I+1,J,K+1)+C(I+1,J+1,K)
VECT 1 ------ DO 3 K=1,32
SCAL 2 |----- DO 2 J=1,32
SCAL 3 ||---- DO 1 I=1,32
     4 |||    A(I+1,J,K)=1/A(I,J,K)
     4 |||    B(I+1,J+1,K)=B(I,J+1,K)+B(I+1,J,K)
```

Figure 2

Vectorization of loop with recurrences: (a) original source program; (b) vectorized program.

(a)

```
      DO 2 K=1,kk
      DO 2 J=1,jj
      S=A(1,J,K)*B(1,J,K)
      DO 1 i=1,ii
    1 S=S+C(I,J,K)/D(I,J,K)
    2 E(1,J,K)=S
```

(b)

```
VECT 1 ------ DO 2 K=1,64
SCAL 2 |----- DO 2 J=1,32
     3 ||     S=A(1,J,K)*B(1,J,K)
SCAL 3 ||---- DO 1 I=1,32
     4 |||    S=S+C(I,J,K)/D(I,J,K)
     3 ||     E(1,J,K)=S

or

SCAL 1 ------ DO 2 K=1,32
VECT 2 |----- DO 2 J=1,64
     3 ||     S=A(1,J,K)*B(1,J,K)
SCAL 3 ||---- DO 1 I=1,32
     4 |||    S=S+C(I,J,K)/D(I,J,K)
     3 ||     E(1,J,K)=S

or

SCAL 1 ------ DO 2 K=1,32
SCAL 2 |----- DO 2 J=1,32
     3 ||     S=A(1,J,K)*B(1,J,K)
VECT 3 ||---- DO 1 I=1,64
     4 |||    S=S+C(I,J,K)/D(I,J,K)
     3 ||     E(1,J,K)=S
```

Figure 3

Change in vectorization as loop size varies: (a) original source program; (b) vectorized program.

six statements is accessed consecutively, and the statements have been merged as much as possible into common loops.

Figure 2 shows how recurrences can affect vectorization. The three statements in this example have recurrences at the inner one, two, and three loops respectively. Since there are only three loops surrounding these statements, the third statement cannot be vectorized. The other two can be vectorized on outer loops.

Figure 3 shows how the vectorization for a nest of loops can change as the loops vary in the number of iterations. Three examples are given; the code is the same in each case except for the loop counts. Note that this loop includes a reduction. The reduction is performed individually and explicitly when the innermost loop is selected for vectorization, and the value of S at the end of each execution of the inner loop reflects a sum along the I dimension. When an outer loop is selected, however, the reduction is performed in parallel. A vector register is used to hold up to 128 different values of S, reflecting 128 different iterations of either the J or the K loop, and the values fetched by the I loop are accumulated into the register. When the I loop is completed then all 128 values of S are stored.

Dubrulle et al. [10] contains an extended discussion and additional examples of the use of the compiler.

Concluding remarks

This paper has described an optimizing vectorizing production compiler. It identifies many new vectorization possibilities, particularly those involving inner-loop recurrences. It selects from these possibilities the one which will result in the fastest execution of the program, and it compiles highly optimized object code for both the scalar and the vector parts of the program. It does all of this quickly.

This project would never have been possible without the fundamental research of Professor Kenneth W. Kennedy and Dr. J. Randal Allen of Rice University. They in turn credit the earlier work of Professor David J. Kuck and Dr. Uptal Banerjee of the University of Illinois. But it is to Ken Kennedy and Randy Allen that we acknowledge our debts. Throughout our project they offered clarity and support. Their initial programs were lucid and elegant statements of dependence theory and source-to-source vectorization. Without their initial push and continuing help this compiler would not have been produced.

References

1. K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *Technical Report 476-029-4*, Rice University, Houston, TX, October 1980.
2. J. R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Department of Mathematical Sciences, Rice University, Houston, TX, 1982.
3. J. R. Allen, K. Kennedy, and J. Warren, "Automatic Loop Interchange," extended abstract, Department of Mathematical Sciences, Rice University, Houston, TX, July 1982.
4. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Conference Record*, Tenth ACM Symposium on Principles of Programming Languages, Austin, TX, January 1983, pp. 177–189.
5. J. R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston, TX, May 1983.

6. D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, "Compiler Transformation of Dependence Graphs," *Conference Record*, Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, VA, January 1981, pp. 207–218.
7. U. Banerjee, "Data Dependence in Ordinary Programs," *Report No. 873*, Department of Computer Science, University of Illinois at Champaign-Urbana, November 1977.
8. U. Banerjee, "Speedup of Ordinary Programs," Ph.D. Thesis, *Report. No. 79-989*, Department of Computer Science, University of Illinois at Champaign-Urbana, October 1979.
9. R. G. Scarborough and H. G. Kolsky, "Improved Optimization of FORTRAN Object Programs," *IBM J. Res. Develop.* **24**, No. 6, 660–676 (November 1980).
10. A. A. Dubrulle, R. G. Scarborough, and H. G. Kolsky, "How to Write Good Vectorizable Fortran," *Technical Report G320-3478*, IBM Scientific Center, Palo Alto, CA, September 1985.

Received August 28, 1985; accepted for publication November 1, 1985

Randolph G. Scarborough *IBM Scientific Center, P.O. Box 10500, Palo Alto, California 94303.* Mr. Scarborough is a senior scientific staff member at the Palo Alto Scientific Center. He has been active in writing compilers and emulators. He joined IBM in 1969 as a systems engineer at the Trenton, New Jersey, branch office. While there he worked on several large scientific accounts and state government accounts. In 1973 he came to the Palo Alto Scientific Center to develop the APL microcode for the System/370 Model 135. In 1978 he produced the Fortran H Extended Optimization Enhancement. In 1983 this work was augmented to include the new extended-precision expanded-exponent (XEXP) number format. Mr. Scarborough received a B.A. from Princeton University, New Jersey, in 1968. He has received many IBM awards, including three IBM Outstanding Innovation Awards and a Corporate Award for his work on Fortran H Extended Optimization Enhancement.

Harwood G. Kolsky *IBM Scientific Center, P.O. Box 10500, Palo Alto, California 94303.* Dr. Kolsky, an IBM Fellow, is a physicist who has been involved in a variety of projects including programming languages, scientific applications, and digital image processing. He is currently on sabbatical at the University of California, Santa Cruz. After seven years at the Los Alamos Scientific Laboratory, he joined IBM in 1957 in Poughkeepsie, New York, as a member of the product planning group for the STRETCH computer. In 1959 he became assistant manager of the IBM Federal Systems Division office in Omaha, Nebraska. Following this he spent some time at FSD headquarters before being named manager of the Systems Science Department of the San Jose Research laboratory in 1961. In 1962 he headed an advanced technology group in the Advanced Systems Development Division at Los Gatos. He joined the Palo Alto Scientific Center when it was formed in 1964 as manager of the atmospheric physics group. In 1968 he was named consultant and in 1969 IBM Fellow. He served on the Corporate Technical Committee at Armonk from 1974 to 1975. Dr. Kolsky received his B.S. degree from the University of Kansas, Lawrence, in 1943 and his Ph.D. in physics from Harvard University, Cambridge, Massachusetts, in 1950. He is a member of the American Physical Society, the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, and Sigma Xi.