Parallel Computing 17 (1991) 1223–1244
North-Holland

A comparative study of automatic vectorizing compilers

David Levine [a,*], David Callahan [b] and Jack Dongarra [c,†]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4801, USA*
[b] *Tera Computer, 400 North 34th Street, Suite 300, Seattle, WA 98103, USA*
[c] *Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301, USA*

Received June 1991

Abstract

Levine, D., D. Callahan and J. Dongarra, A comparative study of automatic vectorizing compilers, Parallel Computing 17 (1991) 1223–1244.

We compare the capabilities of several commercially available, vectorizing Fortran compilers using a test suite of Fortran loops. We present the results of compiling and executing these loops on a variety of supercomputers, mini-supercomputers, and mainframes.

Keywords. FORTRAN loops; automatic vectorizing compilers; speedup; result analysis; compiler performance model.

1. Introduction

This paper describes the use of a collection of Fortran loops to test the analysis capabilities of automatic vectorizing compilers. An automatic vectorizing compiler is one that takes code written in a serial language (usually Fortran) and translates it into vector instructions. The vector instructions may be machine-specific or in a source form such as the proposed Fortran 90 array extensions or as subroutine calls to a vector library.

Most of the loops in the test suite were written by people involved in the development of vectorizing compilers, although several we wrote ourselves. All of the loops test a compiler for a specific feature. These loops reflect constructs whose vectorization ranges from easy to challenging to extremely difficult. We have collected the results from compiling and executing these loops using commercially available, vectorizing Fortran compilers.

The results reported here expand on our earlier work [3]. In that paper, we focused principally on analyzing each compiler's output listing. For the present study, we ran the loops in both scalar and vector modes. In addition, the set of loops has been expanded.

The remainder of this paper is organized into eight sections. Section 2 describes our classification scheme for the loops used in the test. In Section 3 we describe the structure of the test program. In Section 4 we describe the methodology used to perform the test. Section 5 reports on the number of loops that vectorized according to the compiler's output listing. Section 6 presents two aspects of the speedup results. In Section 7 we discuss our model of

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, US Department of Energy, under Contract W-31-109-Eng-38.

† This work was supported in part by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400 and in part by NSF cooperative agreement CCR-8809615.

optimal vector performance and present the results of comparing the actual performance with the model. Section 8 discusses several aspects of the test. In Section 9 we make some remarks about future work.

2. Classification of loops

The objective of the test suite is to test four broad areas of a vectorizing compiler: dependence analysis, vectorization, idiom recognition, and language completeness. All of the loops in this suite are classified into one of these categories.

We define all terms and transformation names but discuss dependence analysis and program transformation only briefly. Recent discussions of these topics can be found in Allen and Kennedy [2], Padua and Wolfe [12], and Wolfe [15]. For a practical exposition of the application of these techniques, see Levesque and Williamson [6].

2.1. Dependence analysis

Dependence analysis comprises two areas: global data-flow analysis and dependence testing. Global data-flow analysis refers to the process of collecting information about array subscripts. Dependence testing refers to the process of testing for memory overlaps between pairs of variables in the context of the global data-flow information.

Dependence analysis is the heart of vectorization, but it can be done with very different levels of sophistication ranging from simple pattern matching to complicated procedures that solve systems of linear equations. Many of the loops in this section test the aggressiveness of the compiler in normalizing subscript expressions into linear form for the purpose of enhanced dependence testing.

1. *Linear dependence testing.* Given a pair of array references whose subscripts are linear functions of the loop control variables that enclose the references, decide whether the two references ever access the same memory location. When the references do interact, additional information can be derived to establish the safety of loop restructuring transformations.
2. *Induction variable recognition.* Recognize auxiliary induction variables (e.g. variables defined by statements such as $K = K + 1$ inside the loop). Once recognized, occurrences of the induction variable can be replaced with expressions involving loop control variables and loop invariant expressions.
3. *Global data-flow analysis.* Collect global (entire subroutine) data-flow information, such as constant propagation or linear relationships among variables, to improve the precision of dependence testing.
4. *Nonlinear dependence testing.* Given a pair of array references whose subscripts are not linear functions, test for the existence of data dependencies and other information.
5. *Interprocedural data-flow analysis.* Use the context of a subroutine in a particular program to improve vectorization. Possibilities include in-line expansion, summary information (e.g. which variables may or must be modified by an external routine), and interprocedural constant propagation.
6. *Control flow.* Test to see whether certain vectorization hazards exist and whether there are implied dependencies of a statement on statements that control its execution.
7. *Symbolics.* Test to see whether subscripts are linear after certain symbolic information is factored out or whether the results of dependence testing do not, in fact, depend on the value of symbolic variables.

2.2. Vectorization

A simple vectorizer would recognize single-statement Fortran DO loops that are equivalent to hardware vector instructions. When this strict syntactic requirement is not satisfied, more sophisticated vectorizers can restructure programs so that it is. Here, program restructuring is divided into two categories: transformations to enhance vectorization and idiom recognition. The first is described here, and the other in the next section.

1. *Statement reordering*. Reorder statements in a loop body to allow vectorization.
2. *Loop distribution*. Split a loop into two or more loops to allow partial vectorization or more effective vectorization.
3. *Loop interchange*. Change the order of loops in a loop nest to allow or improve vectorization. In particular, make a vectorizable outer loop the innermost in the loop nest.
4. *Node splitting*. Break up a statement within a loop to allow (partial) vectorization.
5. *Scalar and array expansion*. Expand a scalar into an array or an array into a higher-dimensional array to allow vectorization and loop distribution.
6. *Scalar renaming*. Rename instances of a scalar variable. Scalar renaming eliminates some interactions that exist only because of reuse of a temporary variable and allows more effective scalar expansion and loop distribution.
7. *Control flow*. Convert forward branching in a loop into masked vector operations; recognize loop invariant IF's (loop unswitching).
8. *Crossing thresholds (index set splitting)*. Allow vectorization by blocking into two sets. For example, vectorize the statement A(I) = A(N − I) by splitting iterations of the I loop into iterations with I less than N/2 and iterations with I greater than N/2.
9. *Loop peeling*. Unroll the first or last iteration of a loop to eliminate anomalies in control flow or attributes of scalar variables.
10. *Diagonals*. Vectorize diagonal accesses (e.g. A(I, I)).
11. *Wavefronts*. Vectorize two-dimensional loops with dependencies in both dimensions by restructuring the loop for diagonal access.

2.3. Idiom recognition

Idiom recognition refers to the identification of particular program forms that have (presumably faster) special implementations.

1. *Reductions*. Compute a scalar value or values from a vector, such as sum reductions, min/max reductions, dot products, and product reductions.
2. *Recurrences*. Identify special first- and second-order recurrences that have logarithmically faster solutions or hardware support.
3. *Search loops*. Search for the first or last instance of a condition, possibly saving index value(s).
4. *Packing*. Scatter or gather a sparse vector from or into a dense vector under the control of a bit mask or an indirection vector.
5. *Loop rerolling*. Vectorize loops where the inner loop has been unrolled.

2.4. Language completeness

This section tests how effectively the compilers understand the complete Fortran language. Simple vectorizers might limit analysis to DO loops containing only floating point and integer assignments. More sophisticated compilers will analyze all loops and vectorize wherever possible.

1. *Loop recognition*. Recognize and vectorize loops formed by backward GO TO's.

2. *Storage classes and equivalencing.* Understand the scope of local vs. common storage; correctly handle equivalencing.
3. *Parameters.* Analyze symbolic named constants, and vectorize statements that refer to them.
4. *Nonlogical IF's.* Vectorize loops containing computed GO TO's and arithmetic IF's.
5. *Intrinsic functions.* Vectorize functions that have elemental (vector) versions such as SIN and COS or known side effects.
6. *Call statements.* Vectorize statements in loops that contain CALL statements or external function invocations.
7. *Nonlocal GO TO's.* Branches out of loops, RETURN statements or STOP statements inside of loops.
8. *Vector semantics.* Load before store, and preserve order of stores.
9. *Indirect addressing.* Vectorize subscripted subscript references (e.g. A(INDEX(I))) as Gather/Scatter.
10. *Statement functions.* Vectorize statements that refer to Fortran statement functions.

3. Test program structure

The test program consists of 122 loops that represent different constructs intended to test the analysis capabilities of a vectorizing compiler. Using the classification scheme in Section 2, there are 29 loops in the Dependence Analysis category, 41 loops in the Vectorization category, 24 loops in the Idiom Recognition category, and 28 loops in the Language Completeness category. Also included are 13 additional 'control' loops we expect all compilers to be able to vectorize. These allow us to measure the rates of certain basic operations for use with the model discussed in Section 7.

The majority of the test loops operate on one-dimensional arrays; a small number operate on two-dimensional arrays. Most of the loops in the test are fairly short; many are a single statement and others usually no more than several statements. Many of the loops access memory with a stride of one. Each loop is contained in a separate subroutine. A driver routine calls each subroutine with vector lengths of 10, 100, and 1000.

```
   subroutine s111 (ntimes,ld,n,ctime,dtime,a,b,c,d,e,aa,bb,cc)
   integer ntimes, ld, n, i, nl
   real a(n), b(n), c(n), d(n), e(n), aa(ld,n), bb(ld,n), cc(ld,n)
   real t1, t2, second, chksum, ctime, dtime, cs1d
   call init(ld,n,a,b,c,d,e,aa,bb,cc, s111 )
   t1=second()
   do 1 nl=1,2*ntimes
   do 10 i=2,n,2
      a(i)=a(i-1)+b(i)
10 continue
   call dummy(ld,n,a,b,c,d,e,aa,bb,cc,1.)
1  continue
   t2=second()-t1-ctime-( dtime * float(2*ntimes) )
   chksum=cs1d(n,a)
   call check (chksum,2*ntimes*(n/2),n,t2, s111 )
   return
   end
```

Fig. 1. Example loop.

V	– vectorized
P	– partially vectorized
N	– not vectorized
V/P	– fully or partially vectorized

Fig. 2. Key to symbols for Tables 1–8, 12–13.

An example loop is shown in *Fig. 1*. Relevant operands are initialized once at the start of the loop. An outer repetition loop is used to increase the granularity of the calculation, thereby avoiding problems with clock resolution. A call to a dummy subroutine is included in each iteration of the repetition loop so that, in cases where the inner loop calculation is invariant with respect to the repetition loop, the compiler is still required to execute each iteration rather than just recognizing that the calculation needs to be done only once.

After execution of the loop is complete, a checksum is computed by using the result array(s). The checksum and the time used are then passed to a check subroutine. The check subroutine verifies the checksum with a precomputed result and prints out the time to execute the loop. The time is calculated by calling a timer at the start of the loop and again at the end of the loop and taking the difference of these times minus the cost of the timing call and the cost of the multiple calls to the dummy subroutine.

4. Test methodology

The test program is distributed in two files: a driver program in one file, and the test loops in the other. The files were distributed to interested vendors, who were asked to compile the loops without making any changes [1] using only the compiler options for automatic vectorization. Thus, the use of compiler directives or interactive compilation features to gain additional vectorization was not tested. Vendors were asked to make two separate runs of the test: one using scalar optimizations only, and the other using the same scalar optimizations and, in addition, all automatic vectorization options. Vendors with multiprocessor computers submitted uniprocessor results only. Appendix A contains details of the exact machine configurations and versions of the software used.

The rules require separate compilation of the two files. The rules for compilation of the driver file require that no compiler optimizations be used and that the file not be analyzed interprocedurally to gather information useful in optimizing the test loops.

The file containing the loops was compiled twice – once for the scalar run and once for the vector run. For the scalar run, global (scalar) optimizations were used. For the vector run, in addition to the same global optimizations specified in the scalar run, vectorization and – if available – automatic call generation to optimized library routines, function inlining, and interprocedural analysis were used.

All files were compiled to use 64-bit arithmetic. Most runs were made on standalone systems. [2] For virtual memory computers, the runs were made with a physical memory and working-set size large enough that any performance degradation from page faults was negligible. In all cases the times reported to us were *user* CPU time.

After compiling and executing the loops, the vendors sent back the compiler's output listing (source echo, diagnostics, and messages) and the output of both the scalar and vector runs. We

[1] One vendor was allowed to (1) separate a 135-way IF-THEN-ELSEIF-ELSE construct in order to overcome a self-imposed limit, and (2) include the array declarations in a common block in the driver program (only) in order to overcome a self-imposed limit on memory allocation size. Neither modification had any impact on performance.

[2] The CRAY Computer and Hitachi runs were not.

then examined the compiler's output listings to see which loops had been vectorized, and analyzed the scalar and vector results. In addition to measuring the execution time of the loops, we checked the numerical result in order to verify correctness. However, the check was strictly for correctness of the numerical result; no attempt was made to see whether possibly unsafe transformations had been used.

5. Number of loops vectorized

In this section we discuss the number of loops that were vectorized, as reported by the compiler's output listing. All of the loops in our test are amenable to some degree of vectorization. For some loops, this may only be partial vectorization; for others, vectorization may require the use of optimized library routines or special hardware.

5.1. Definition of vectorization

We define a *statements* as vectorizable if one or more of the *expressions* in the statement involve array references or may be converted to that form. We define three possible results for a compiler attempting to vectorize a loop. A *loop* is *vectorized* if the compiler generates vector instructions for all vectorizable statements in the loop. A loop is *partially vectorized* if the compiler generates vector instructions for some, but not all, vectorizable statements in the loop. No threshold is defined for what percentage of a loop needs to be vectorized to be listed in this category, only that some expression in a statement in the loop is vectorized. A loop is *not vectorized* if the compiler does not generate vector instructions for any vectorizable statements within the loop.

For some loops the Cray Research, FPS Computing, IBM, and NEC compilers generated a runtime IF-THEN-ELSE test which executed either a scalar loop or a vectorized loop. These loops have been scored as either *vectorized* or *not vectorized* according to whether or not vectorized code was actually executed at runtime.

The Cray Computer compiler 'conditionally vectorized' certain loops. That is, for loops with ambiguous subscripts, a runtime test was compiled that selected a safe vector length.[3] These loops have been scored as either *vectorized* if the safe vector length was greater than one, otherwise not *vectorized*.

For a number of loops, the Fujitsu compiler generated scalar code even though the compiler indicated that partial vector code could be generated. In these cases, the compiler listing contained the message "Partial vectorization overhead is too large", indicating that although partial vectorization was possible, for these loops the compiler considered scalar code more efficient. These loops have been scored as *partially vectorized*.

Our definition of vectorization counts as vectorized those loops that are recognized by the compiler and *automatically* replaced by calls to optimized library routines. In some cases a compiler may generate a call to an optimized library routine rather than explicitly generating vector code. Typical examples are for certain reduction and recurrence loops. Often the library routines use a mix of scalar and vector instructions; while perhaps not as fast as pure vector loops, since the construct itself is not fully parallel, they are usually faster than scalar execution. In all cases where the compiler automatically generated a call to a library routine, we have scored the loop as vectorized.

[3] A safe vector length is one that allows the compiler to execute vector instructions and still produce the correct result. For example, the statement A(I) = A(I-7) with loop increment one may be executed in vector mode with any vector length less than or equal to 7.

Table 1
Full vectorization (122 loops)

Compute	V	P	N
CONVEX C210	68.0 (83)	10.7 (13)	21.3 (26)
CCC CRAY-2	60.7 (74)	1.6 (2)	37.7 (46)
CRI CRAY Y-MP	77.9 (95)	8.2 (10)	13.9 (17)
DEC VAX 9000-210	60.7 (74)	3.3 (4)	36.1 (44)
FPS M511EA-2	72.1 (88)	4.9 (6)	23.0 (28)
Fujitsu VP2600/10	71.3 (87)	16.4 (20)	12.3 (15)
Hitachi S-820/80	71.3 (87)	7.4 (9)	21.3 (26)
IBM 3090-600J	77.9 (95)	4.9 (6)	17.2 (21)
NEC SX-X/14	72.1 (88)	5.7 (7)	22.1 (27)
Average	70.2 (85)	7.0 (8)	22.8 (27)

5.2. Results

Tables 1–6 list the results of analyzing the compilers' listings. Each table contains the percentage of loops in each column, followed by the actual number in parentheses. *Table 1* summarizes the results for all 122 loops. *Table 2* is also a summary of all the loops; here, however, the column V/P counts the loops that were either fully or partially vectorized. *Tables 3–6* contain results by category as defined in Section 2.

5.3. Analysis of results

The average number of loops vectorized was 70%, and vectorized or partially vectorized was 77%. The best results were 78% and 88%, respectively. Of the 122 loops, only two were not vectorized or partially vectorized by any of the compilers; both loops are vectorizable. There is probably no significant difference between compilers within a few percent of each other. Slight differences may be due to different hardware, the availability of special software libraries, the architecture of a machine being better suited to executing scalar or parallel code for certain constructs, or the makeup of the loops used in our test.

From *Table 1* we see that the Cray Research and IBM compilers vectorized the most loops. A large number of other compilers are grouped closely together and only a few loops behind these two. Comparing *Table 1* to *Table 2*, we see that counting partially vectorized loops in the totals allows the Fujitsu compiler to vectorize the most loops. It is interesting to note, however,

Table 2
Full and partial vectorization (122 loops)

Computer	V/P	N
CONVEX C210	78.7 (96)	21.3 (26)
CCC CRAY-2	62.3 (76)	37.7 (46)
CRI CRAY Y-MP	86.1 (105)	13.9 (17)
DEC VAX 9000-210	63.9 (78)	36.1 (44)
FPS M511EA-2	77.0 (94)	23.0 (28)
Fujitsu VP2600/10	87.7 (107)	12.3 (15)
Hitachi S-820/80	78.7 (96)	21.3 (26)
IBM 3090-600J	82.8 (101)	17.2 (21)
NEC SX-X/14	77.9 (95)	22.1 (27)
Average	77.2 (94)	22.8 (27)

Table 3
Dependence analysis (29 loops)

Computer	V	P	N
CONVEX C210	65.5 (19)	17.2 (5)	17.2 (5)
CCC CRAY-2	69.0 (20)	0.0 (0)	31.0 (9)
CRI CRAY Y-MP	86.2 (25)	0.0 (0)	13.8 (4)
DEC VAX 9000-210	69.0 (20)	0.0 (0)	31.0 (9)
FPS M511EA-2	82.8 (24)	0.0 (0)	17.2 (5)
Fujitsu VP2600/10	65.5 (19)	24.1 (7)	10.3 (3)
Hitachi S-820/80	55.2 (16)	10.3 (3)	34.5 (10)
IBM 3090-600J	86.2 (25)	0.0 (0)	13.8 (4)
NEC SX-X/14	75.9 (22)	6.9 (2)	17.2 (5)
Average	72.8 (21)	6.5 (1)	20.7 (6)

Table 4
Vectorization (41 loops)

Computer	V	P	N
CONVEX C210	73.2 (30)	14.6 (6)	12.2 (5)
CCC CRAY-2	34.1 (14)	4.9 (2)	61.0 (25)
CRI CRAY Y-MP	56.1 (23)	22.0 (9)	22.0 (9)
DEC VAX 9000-210	58.5 (24)	7.3 (3)	34.1 (14)
FPS M511EA-2	61.0 (25)	14.6 (6)	24.4 (10)
Fujitsu VP2600/10	68.3 (28)	24.4 (10)	7.3 (3)
Hitachi S-820/80	78.0 (32)	9.8 (4)	12.2 (5)
IBM 3090-600J	75.6 (31)	12.2 (5)	12.2 (5)
NEC SX-X/14	65.9 (27)	12.2 (5)	22.0 (9)
Average	63.4 (26)	13.6 (5)	23.0 (9)

Table 5
Idiom recognition (24 loops)

Computer	V	P	N
CONVEX C210	66.7 (16)	4.2 (1)	29.2 (7)
CCC CRAY-2	70.8 (17)	0.0 (0)	29.2 (7)
CRI CRAY Y-MP	87.5 (21)	4.2 (1)	8.3 (2)
DEC VAX 9000-210	54.2 (13)	4.2 (1)	41.7 (10)
FPS M511EA-2	70.8 (17)	0.0 (0)	29.2 (7)
Fujitsu VP2600/10	87.5 (21)	8.3 (2)	4.2 (1)
Hitachi S-820/80	91.7 (22)	4.2 (1)	4.2 (1)
IBM 3090-600J	58.3 (14)	0.0 (0)	41.7 (10)
NEC SX-X/14	87.5 (21)	0.0 (0)	12.5 (3)
Average	75.0 (18)	2.8 (0)	22.2 (5)

that of the 20 loops we counted as partially vectorized by the Fujitsu compiler, only two actually resulted in (partial) vector code being executed at runtime. For the other 18 loops the Fujitsu compiler made the decision that it would not be cost effective to partially vectorize them. The Convex compiler also did a significant amount of partial vectorization.

Tables 3–6 show that some compilers did particularly well in certain categories. The Cray Research, FPS Computing, and IBM compilers had the best results in the Dependence Analysis category. The Convex, Hitachi, and IBM compilers had the best results in the Vectorization

Table 6
Language completeness (28 loops)

Computer	V	P	N
CONVEX C210	64.3 (18)	3.6 (1)	32.1 (9)
CCC CRAY-2	82.1 (23)	0.0 (0)	17.9 (5)
CRI CRAY Y-MP	92.9 (26)	0.0 (0)	7.1 (2)
DEC VAX 9000-210	60.7 (17)	0.0 (0)	39.3 (11)
FPS M511EA-2	78.6 (22)	0.0 (0)	21.4 (6)
Fujitsu VP2600/10	67.9 (19)	3.6 (1)	28.6 (8)
Hitachi S-820/80	60.7 (17)	3.6 (1)	35.7 (10)
IBM 3090-600J	89.3 (25)	3.6 (1)	7.1 (2)
NEC SX-X/14	64.3 (18)	0.0 (0)	35.7 (10)
Average	73.4 (20)	1.6 (0)	25.0 (7)

category. The Cray Research, Fujitsu, Hitachi, and NEC compilers had the best results in the Idiom Recognition category. In the Language Completeness category the Cray Research and IBM compilers had the best results. The Vectorization category seemed the most difficult, with approximately 10% fewer loops vectorized overall than for the other sections.

Certain sections seemed fairly easy, with most vendors vectorizing or partially vectorizing almost all of the loops. Using the classification scheme of Section 2 these sections were linear dependence testing, global data-flow analysis, statement reordering, loop distribution, node splitting, scalar renaming, control flow, diagonals, loop rerolling, parameters, intrinsic functions, indirect addressing, and statement functions.

In some sections, while many vendors vectorized or partially vectorized most loops, various individual vendors did not do particularly well. These sections were induction variable recognition, interprocedural data-flow analysis, symbolics, scalar and array expansion, reductions, search loops, packing, and nonlogical IF's.

Some sections were difficult for many compilers. Typically, at least half the vendors missed at least some, and sometimes most, of the loops in these sections. These sections were control flow, loop interchange, index set splitting, loop peeling, recurrences, loop recognition, storage classes and equivalencing, and nonlocal GO TO's.

A few sections were particularly difficult, with only one or two compilers doing any vectorization at all. These sections were nonlinear dependence testing, wavefronts, and call statements.

We found that some vendors with approximately equal results did much better in one section than another. Certain induction variable tests, interprocedural data-flow analysis, loop interchange, recurrences, loop recognition, storage classes and equivalence statements, and loops with exits were the sections that showed the greatest variation. We conclude that the compiler vendors have focused their efforts on particular subsets of the features tested by the suite. Possible reasons might include hardware differences or (self-imposed) limits on compilation time, compilation memory use, or the size of the generated code.

Complete results, on a loop-by-loop basis, may be found in [7].

6. Speedup

The goal of vectorization is for the vectorized program to execute in less time than the unvectorized program. The metric used is the speedup, s_p, defined as $s_p = t_s/t_v$, where t_s is the scalar time and t_v is the vector time. In this section we look at two aspects of speedup. First, does the vector code run slower than the corresponding scalar code? Second, how large a speedup can be gained with vectorization?

Table 7
Loops vectorized ($s_p > 0.95$, vector length = 100, 122 loops)

Computer	V	P	N
CONVEX C210	68.0 (83)	9.0 (11)	23.0 (28)
CCC CRAY-2	60.7 (74)	0.8 (1)	38.5 (47)
CRI CRAY Y-MP	77.9 (95)	7.4 (9)	14.8 (18)
DEC VAX 9000-210	49.2 (60)	2.5 (3)	48.4 (59)
FPS M511EA-2	71.3 (87)	3.3 (4)	25.4 (31)
Fujitsu VP2600/10	68.9 (84)	13.1 (16)	18.0 (22)
Hitachi S-820/80	69.7 (85)	1.6 (2)	28.7 (35)
IBM 3090-600J	71.3 (87)	4.1 (5)	24.6 (30)
NEC SX-X/14	72.1 (88)	2.5 (3)	25.4 (31)
Average	67.7 (82)	4.9 (6)	27.4 (33)

Table 8
Loops vectorized ($s_p > 0.95$, vector length = 1000, 122 loops)

Computer	V	P	N
CONVEX C210	68.0 (83)	8.2 (10)	23.8 (29)
CCC CRAY-2	60.7 (74)	0.8 (1)	38.5 (47)
CRI CRAY Y-MP	77.9 (95)	7.4 (9)	14.8 (18)
DEC VAX 9000-210	54.9 (67)	2.5 (3)	42.6 (52)
FPS M511EA-2	71.3 (87)	4.1 (5)	24.6 (30)
Fujitsu VP2600/10	69.7 (85)	15.6 (19)	14.8 (18)
Hitachi S-820/80	70.5 (86)	1.6 (2)	27.9 (34)
IBM 3090-600J	73.0 (89)	4.1 (5)	23.0 (28)
NEC SX-X/14	72.1 (88)	2.5 (3)	25.4 (31)
Average	68.7 (83)	5.2 (6)	26.1 (31)

6.1. Vectorized loops revisted

Ideally the speedup from vectorization (or partial vectorization) should be as large as possible. At a minimum, though the vector code should run at least as fast as the scalar code. However, this minimum is not always achieved, particularly at short vector lengths where there may not be enough work in the loop to overcome the vector startup cost.

Tables 7 and *8* revisit the results in *Table 1*. The number of loops in each of the different categories is again taken from the compiler listing. In *Tables 7* and *8* however, we have not counted as vectorized or partially vectorized any loops where $s_p < 0.95$. [4] The results in *Table 7* are for vector length 100, and the results in *Table 8* are for vector length 1000. We have not presented these results for vector length ten since almost all vendors suffer some performance degradation for short vectors.

The results in *Tables 7* and *8* are mostly consistent with *Table 1*. Four of the compilers show no degradation on any of the vectorized loops. Three others show a degradation on only one or two loops. Only two compilers show a degradation on any significant number of loops. The results for partial vectorization are also fairly consistent with *Table 1*, with only one compiler showing any serious number of loops being degraded. There is a large variance in the test suite as to which loops have degraded performance. No particular trend is obvious.

Two compilers also suffered noticeable performance degradations (below 90%) for a significant number of loops (10 or more) that were *not vectorized*. We believe somehow that the

[4] We use 0.95 instead of 1 to allow for the possibility of measurement error.

Table 9
Aggregate speedup results by section (122 loops)

Section	10		100		1000		All VL	
Data dependence	1.44	0.90	5.09	1.87	12.25	2.15	6.26	1.42
Vectorization	1.37	0.97	5.11	1.74	21.32	1.91	9.27	1.41
Idiom recognition	0.95	0.73	3.66	1.73	11.52	2.29	5.38	1.26
Language completeness	1.55	1.07	4.72	1.90	9.63	2.13	5.30	1.55
All sections	1.35	0.92	4.73	1.80	14.55	2.08	8.61	1.41

attempt to vectorize interfered with the generation of good scalar code. We view this as a performance bug and have advised the vendors. Other than these cases, the vectorizers rarely generated code that was inferior to the scalar code on vector lengths of 100 or more. An exception is the nine loops the CRAY-2 compiler-generated vector code for with a safe vector length of one. These loops, although scored as *not vectorized*, had vector execution times that were frequently twice the scalar execution times.

6.2. Aggregate speedup results

The speedup that can be achieved on a particular vector computer depends on several factors: the speed of the vector hardware relative to the speed of the scalar hardware, the inherent vector parallelism in the code of interest, and the sophistication of the compiler in detecting opportunities to generate code to run on the vector hardware. From the perspective of our test, we would like to measure the speedup achieved just from the compiler's vectorization capabilities. However, speedups are too strongly influenced by architecture and implementation to be meaningful indicators of compiler performance. Therefore, we prefer not to give speedup results for individual vendors which may be misinterpreted as representing compiler performance *only*. Instead, we present speedup statistics using the aggregate results from all vendors.

Table 9 presents a summary of the speedup results of all vendors. The first four rows present results according to the classification scheme in Section 2. Results are given for vector lengths of 10, 100, and 1000 and, in the last column, the sum over all three vector lengths. Each column contains the arithmetic and harmonic means of the speedups for the loops in that section. The results in the last row are summed over all four sections.

Table 10 contains aggregate statistics for three different levels of vectorization. The format of the table is similar to *Table 9*. The first row contains speedup statistics for the 771 loops scored as fully vectorized. The second row contains speedup statistics for the 848 loops scored as either fully or partially vectorized. The last row contains speedup statistics for the 77 loops that were partially vectorized.

6.3. Discussion of speedup

As might be expected, at the relatively short vector length of 10, the speedups were not very large. This is particularly true of the Idiom Recognition section, where the methods used to

Table 10
Aggregate speedup results by type of vectorization

Vectorization level	10		100		1000		All VL	
Full vectorization	1.54	0.96	6.29	3.11	20.20	4.48	9.34	1.90
Full or partial vectorization	1.47	0.93	5.85	2.61	18.56	3.48	8.63	1.72
Partial vectorization	0.83	0.68	1.41	1.00	2.21	1.07	1.49	0.88

vectorize some of the loops are not amenable to the full speedup that can be provided by the hardware. At vector length 100 most speedups were between three and six. At the longest vector length, 1000, the individual speedups were slightly higher for most. Three vendors however, had very large average speedups (29.4, 33.2, and 39.8) over the scalar speed.

The choice of mean clearly affects the results. In the Vectorization section, the arithmetic mean at vector length 1000 is 21.32, while the harmonic mean is only 1.91. These results show that a relatively small number of large speedups can greatly affect the arithmetic mean. McMahon [11] and Smith [13] discuss the different means.

If we compare the last row of *Table 9* with the first two rows in *Table 10*, we see better speedups at all vector lengths when we consider only the loops fully or partially vectorized. Of course which loops were included, and how many, varies for each vendor.

In several loops in the test suite, not all statements can be vectorized. A compiler can still improve performance by partial vectorization – vectorizing some, but not all, of the statements. As *Table 10* shows, the speedups from partial vectorization are significantly less than those from full vectorization. There are several reasons for this result. First, since by definition partial vectorization vectorizes only some of the statements in a loop, others still run at scalar speeds. Second, our definition of partial vectorization classifies as such a loop that uses any vector instructions, no matter how much of the loop is executed in scalar mode. Finally, many techniques for partial vectorization introduce extra work, such as extra loads and stores and additional loop overhead, which is not required in the original loop.

Even with these caveats we see from the last row in *Table 10* that there is still a benefit to be gained from partial vectorization, but primarily at the longer vector lengths. Even more so than with full vectorization, partial vectorization – at least on the test loops – degrades performance at vector length of ten.

7. Percent vectorization

In this section we focus on the performance of the compiler independent of the computer archtecture. We do this by developing a machine-specific model of what optimal vector performance is for each of the loops in our test suite. We then compare the optimal performance predicted by this model with the actual vector execution results to determine the percent of the optimal vector performance actually achieved.

7.1. A model of compiler performance

A simple model of vector performance as a function of vector length is given by the formula [9]

$$t = t_0 + nt_e, \tag{1}$$

where t is the time to execute a vector loop of length n, t_0 is the vector startup time, and t_e is the time to execute a particular vector element. Equivalent to (1) is the well-known model of Hockney (see Hockney and Jesshope [5]),

$$t = r_\infty^{-1}(n + n_{1/2}), \tag{2}$$

where r_∞ is the asymptotic performance rate and $n_{1/2}$ is the vector length necessary to achieve one half the asymptotic rate. Equations (1) and (2) can be shown to be equivalent if we use the definitions $r_\infty = t_e^{-1}$ and $n_{1/2} = t_0/t_e$ [5].

As Lubeck [8] points out, neither equation models the stripmining process used by compilers on register-to-register vector computers. Also, (1) and (2) may not reflect the behavior of

Table 11
Basic operation classes

Class	Operation
0	Load
0	Gather (Load indirect)
1	Store
1	Scatter (Store indirect)
2	Arithmetic (Add, Multiply)
2	Reductions

cache-based machines under increasing vector lengths (see, for example [1]). Nevertheless, for the purposes of our model we believe (1) and (2) to be sufficient.

By analogy with r_∞, for each loop, we define three rates: r_s for the optimized scalar code, r_v for the vector code, and r_o for optimal vector code for the target machine. These rates are defined in units of the number of iterations per second of the loop. We assume $r_s < r_o$, and we expect $r_s \leqslant r_v \leqslant r_o$, although (as the previous section indicated) it is possible to have $r_v < r_s$.

Using the scalar and vector data collected, we can solve (2), for each loop, for r_s and r_v, respectively. Since we cannot necessarily assume $r_v = r_o$, we must *estimate* r_o. To do this, we assume that the execution time of a loop is determined by the basic operations in the loop. To determine the rate at which basic operations (e.g. addition or load) can be performed, we use the control loops, which we assume *can* be optimally vectorized.

We divide the basic operations into classes. Each class contains operations that utilize a specific functional unit. For example, *Table 11* lists the basic operations in each class for a generic computer with separate load and store pipes. [5]

The list of which operations belong to which classes varies by vendor, primarily with respect to the memory operations. For example, on a machine with separate load and store pipes, the load and gather operations are in one class (they compete for the load pipe), and the store and scatter operations are in another class (they compete for the store pipe). For machines with only one pipe for all memory accesses the four memory operations are all in the same class. Even though these operations all have their own execution rates, when they compete for the same resources they are in the same class.

To model control flow, we assume an 'execute under mask' model in which every operation is assumed to be executed in vector mode, and the results of various control paths are merged together. Alternative strategies are possible, such as using compress and expand to perform arithmetic only where selected, but we found that execute under mask was sufficient for our purposes.

On each computer, and for each loop L, we estimate its optimal execution rate r_o, using the algorithm shown in *Fig. 3.* Here C represents the set of classes defined for a particular computer, o the operations in a class, N_o the number of instances of o in L, and R_o the rate for operation o (in units of operations per second) measured with the control loops. The algorithm assumes that operations in different classes execute concurrently while operations in the same class execute sequentially.

[5] This table could be extended by subdividing classes into special cases. For example, the arithmetic class could be divided into separate addition and multiplication classes. For machines that can execute adds and multiplies concurrently – all machines in this study – these multiple functional units are modeled as simply a higher arithmetic-processing rate. The difference in execution times between computing the elementwise sum of three vectors and the elementwise product was insignificant for all computers. This fact is not surprising, since the rate limiting step for almost all loops in the suite is memory references, and so this distinction would not change our results significantly.

```
maxtime ← 0
foreach c ∈ C occurring in L
    time ← 0
    foreach o ∈ c
        time ← time + $N_o / R_o$
    endfor
    maxtime ← max(time,maxtime)
endfor
$r_o$ ← 1 / maxtime
```

Fig. 3. Algorithm for estimating optimal execution rate.

This model is based on the notion of a resource limit, similar to the model used to calculate performance bounds in [10,14]. We assume that for each loop there exists a particular class of operations that use the same function unit and that the time to execute these operations provides a lower bound on the time to execute the loop. The algorithm in Figure 3 calculates that bound, and we use its reciprocal as r_o.

In addition to measuring the basic vector operation rates, we also measure the basic scalar operation rates. For each loop, we then determine which operations can be executed in vector mode and which must be executed in scalar mode. We then modify the algorithm in *Fig. 3* to use the appropriate rate (vector or scalar) for R_o for each operation.

For each loop and each vendor, we have now determined the three execution rates: r_o using the algorithm given in *Fig. 3*, and r_s and r_v using (2). All three rates were computed by using the data for vector lengths of 100 and 1000. We now define *percent vectorization*, p_v, by the formula

$$p_v = \frac{r_v - r_s}{r_o - r_s}. \tag{3}$$

With this definition, if a loop's vector execution rate is the same as the scalar rate, $p_v = 0\%$, and if a loop's vector execution rate is the same as the optimal vector execution rate, $p_v = 100\%$. We can now classify a loop as vectorized, partially vectorized, or not vectorized according to the value of p_v. We do this according to the rule

$$\text{Result} = \begin{cases} n & p_v < 10\% \\ p & 10\% \leqslant p_v < 50\% \\ v & 50\% \leqslant p_v. \end{cases} \tag{4}$$

7.2. Example

In this section we show an example of the computation of p_v for two computers, C_1 and C_2. We assume that C_1 has two load pipes and a store pipe and that C_2 has one pipe used for both loads and stores.

The example used is the loop shown in *Fig. 1*. For this loop we have the following profile of basic operation, N_0:

Load	Store	Gather	Scatter	Arithmetic	Reductions
0 2	0 1	0 0	0 0	0 1	0 0

The first number in each pair is the number of scalar operations, and the second is the number of vectorizable operations. In this example, executing the loop requires two vector loads, one vector store, and a vector addition. No scalar operations are required (our model takes into account scalar operations that occur within the loop body, but not scalar operations,

Table 12
Full vectorization according to (3) and (4) (122 loops)

Computer	V	P	N
CONVEX C210	51.6 (63)	14.8 (18)	33.6 (41)
CCC CRAY-2	36.1 (44)	24.6 (30)	39.3 (48)
CRI CRAY Y-MP	54.1 (66)	25.4 (31)	20.5 (25)
DEC VAX 9000-210	45.1 (55)	8.2 (10)	46.7 (57)
FPS M511EA-2	52.5 (64)	18.0 (22)	29.5 (36)
Fujitsu VP2600/10	53.3 (65)	13.1 (16)	33.6 (41)
Hitachi S-820/80	52.5 (64)	15.6 (19)	32.0 (39)
IBM 3090-600J	51.6 (63)	18.9 (23)	29.5 (36)
NEC SX-X/14	46.7 (57)	21.3 (26)	32.0 (39)
Average	49.3 (60)	17.8 (21)	33.0 (40)

such as incrementing the loop control variable or testing for loop termination, that have to do with the loop control itself).

Using the results of the control loops, we have calculated the following basic vector operation rates. The units are in million of operations per second.

Computer	Load	Store	Arithmetic
C_1	227	150	269
C_2	186	207	286

Using these values and the loop profile above, we can estimate r_o with the algorithm shown in *Fig. 3*. The result of these calculations is that, for C_1, the optimal vector execution rate is 114 million iterations per second, and for C_2 it is 64 million iterations per second.

Using the scalar and vector results for vector lengths 100 and 1000, we determined the follwing results for r_s and r_v by solving (2):

Computer	r_s	r_v
C_1	12.3	115.0
C_2	10.7	19.8

Substituting the appropriate values of r_s, r_v and r_o, into (3), we calculated $p_v = 100\%$ for C_1 and $p_v = 17\%$ for C_2. Applying (4), we determined that C_1 fully vectorizes this loop and that C_2 partially vectorizes this loop.

7.3. *Results*

Table 12 is similar to *Table 1*, except here the number of loops vectorized or partially vectorized has been determined by applying (3) and (4) as opposed to analyzing the compiler's output listing.

In comparing *Table 12* to *Table 1* we observe that the results are mostly consistent with *Table 1*, with a somewhat tighter grouping among vendors with the most loops vectorized. Most compilers vectorized between 20 and 30 loops which did not acheive full vector performance ($p_v \geqslant 50\%$).

Casual inspection of the data indicates that there are a number of loops for which at most one vendor successfully achieved vector performance and all other vendors that vectorized did not. Approximately 23 loops account for most of the differences between the two measures of vectorization. For the most part, these loops are scattered across categories but they include most of the scalar expansion loops, the search loops, the packing loops, and the loops with multiway branching.

Factors other than simple detection of vectorizablility are reflected in the computation of vectorization percentages. In particular, traditional optimizations such as common subexpression elimination, register allocation, and instruction scheduling will all influence the quality of the generated code and hence the percentage of vectorization. In this sense, the percentage is more a measure of the overall quality of the compiler generated code.

Optimal code generation and, in particular, instruction scheduling for very simple loops are extremely difficult. For loops with large bodies, heuristic algorithms will usually get within a small number of instructions of what is optimal. When the loop body contains only five or ten instructions, however, being off by a 'small' number could cost 25% of achievable performance. Thus, since almost all of the loops in the suite are very simple, the compilers may perform substantially better on 'real' codes than is suggested by *Table 12*.

That the measured execution rates are lower than what might be expected from 'vector' code may be due to model limitations. For example, the model treats unit and nonunit stride vector accesses as equal in cost: there was no convincing evidence that nonunit stride was a factor worth adding to the operation classes listed in *Table 11*. The other major factor not modeled is the presence of a data cache, its size and its organization. This is discussed in Section 8.3.1.

One issue that biases the results presented here is that we used the measured performance on simple loops to calibrate the model. Thus our 'optimal rates' may be significantly below 'machine peaks' since those peaks may be achievable only assuming optimal compilation. Further, if the code generation capabilities of one compiler are generally poor compared with another, then its ability to vectorize may appear inflated, since our estimate of optimal execution rate may be too low. This situation can be corrected by replacing the control loops with numbers derived from hand-crafted assembly routines that would provide estimates of 'achievable peaks'. We did not have the resources to generate these numbers for each machine.

8. Discussion

8.1. Validity of the test suite

How good is this test suite? The question can be answered in several ways, but we will address three specific areas: coverage, stress, and accuracy.

8.1.1 Coverage

By 'coverage' we refer to how well the test suite represents typical, common, or important Fortran programming practices. We would like to assert that high effectiveness on the test suite will correspond to high effectiveness in general. Unfortunately, there is no accepted suite of Fortran programs that can be called representative, and so we have no quantitative way of determining the coverage of our suite. We believe, however, that the method used to select the tests has yielded reasonable coverage. This method consisted of two phases.

In the first phase, a large number of loops were collected from several vendors and interested parties. This gave a diverse set of viewpoints, each with a different machine architecture and hence somewhat different priorities. In some cases the loops represented 'real' code from programs that had been benchmarked. The majority, however, were specifically written to test a vectorizing compiler for a particular feature. Independently, the categorization scheme used in Section 2 was developed based on experience and on published literature about vectorization.

In the second phase, the test suite was culled from the collected loops by classifying each loop into one or more categories and then selecting a few representative loops from each category. Our interest was in coverage; and since 'representative' is not well defined, we made no attempt to weight some of the subcategories more than others by changing the number of

Table 13
Loops sorted by difficulty

```
[illegible]
```

loops. Where we felt that testing a subcategory required a range of situations, we included several loops; in other cases we felt that one or two loops sufficed. There is significant weighting between major categories. For example, the test suite places greater emphasis on basic vectorization (41 loops) than on idiom recognition (24 loops). This weighting was an artifact of the selected categories and was reflected in the original collection of samples. We felt that this weighting was reasonable and made no attempt to adjust it.

8.1.2 Stress

By 'stress' we refer to how effectively the test suite tests the limits of the compilers. We wish the test to be difficult but not impossible. Again there is no absolute metric against which we can measure the test suite, but we can use the performance of the compilers as a measure. *Table 13* lists the results for the various compilers. In this table, each row corresponds to a particular compiler. Rows are sorted in order of decreasing full and partial vectorization (see *Table 2*). Each column corresponds to a particular loop, and the columns are sorted in order of increasing difficulty.

The loop scores at the bottom of *Table 13* are based on the number of compilers that vectorized or partially vectorized the loop. Many of the loops are inherently only partially vectorizable, and so we have not attempted to weight full versus partial vectorization. We interpret a low score as an indication of a difficult test. From the table we observe a skewed distribution of results, with many of the loops 'easy' (everyone vectorizes) and only a few 'difficult' (only one or two vendors even partially vectorizes).

Viewed from a historical perspective, the test appears less stressful now than it did originally. We can see this qualitatively from *Table 14*, which is reprinted from [3]. Here there seems to be a more balanced distribution of tests between 'easy' and 'difficult' when compared

Table 14
Loops sorted by difficulty, from [3]

```
[illegible]
```

to *Table 13*. Statistics also support this view. In [3] the average number of loops vectorized was 55%, and vectorized or partially vectorized was 61%. Even if we restrict ourselves to just the eight vendors also participating in this test, the previous results are still only 59% and 64%, respectively In this test the average number of loops vectorized was 70%, and vectorized or partially vectorized was 77%, an improvement of about 15%

Several factors may be at work. First, compilers have evolved and improved over time. Second, specialized third-party compiler technology is now readily available to interested vendors. Third, for various reasons approximately half the vendors who participated in the previous test did not participate this time. While those who did not participate span the spectrum of previous results, most had results in the lower or middle part of the previous test. While we added new loops to this test (and also deleted a small number), this does not seem to have provided adequate stress. Since one valid use of this test suite is for compiler writers to diagnose system deficiencies, we expect over time that the test will lose its effectiveness to stress compilers.

8.1.3 Accuracy

By 'accuracy' we refer to how well the test can measure the quality of a vectorizing compiler. Since the difficulty of the tests was determined by the performance of the compilers, it would be circular now to judge the absolute quality of the compilers by their performance on this suite. What about relative performance? It is tempting to distill the results for each compiler into a single number and use that to compare the systems. Such an approach, however, is clearly incorrect, since these compilers cannot be compared in isolation from the machine environment and target application area for which they were designed.

We conclude that the suite represents reasonable coverage, that the stress may no longer be adequate, and that we cannot determine the accuracy of the suite.

8.2. Beating the test

Some of the loops were vectorized in ways that defeated the intent of the test. One example is the use of a runtime test. If the compiler cannot determine at compile time whether a loop is safe to vectorize, because of, say, an unknown parameter value, it must either not vectorize the loop or else generate an alternative code runtime test. At runtime, based on the value of the unknown parameter, the test executes either a scalar or a vector version of the loop, as appropriate. In general, we view runtime testing as a good thing to do. It allows vectorization of loops that would not otherwise be vectorized and allows cost-effectiveness decisions to be deferred until runtime. However, it has a negative side. First, the cost of the test is incurred each time the loop is executed. Second, for large loop nests, it is possible to have a combinatorial explosion in the number of tests generated. All of the loops in our test suite can be determined to be vectorizable at compile time, and thus runtime testing is not necessary. The Cray Research, FPS, IBM, and NEC compilers, however, can generate runtime tests and in a few cases were able to 'beat the test' this way.

A technique similar to runtime testing is conditional vectorization, which was used by the Cray Computer compiler. With conditional vectorization, a safe vector length [6] is calculated at runtime. While conditional vectorization is also good for a compiler to be able to do, it also has a negative side. First, there is the overhead involved in calculating the safe vector length at runtime. Second, if the calculated safe vector length is one, it is more efficient to execute a scalar instruction rather than a vector instruction. None of the loops in our test require

[6] See Section 5.1.

conditional vectorization. Nevertheless, the Cray Computer compiler conditionally vectorized 20 loops, 11 of which resulted in a safe vector length greater than one.

Another way compilers defeated the intent of the test was by their ability to vectorize recurrences, using either library routines or special hardware. Several of the tests call for the compiler to split up a loop (loop distribution, node splitting) or change the order of a loop nest (loop interchange) in order to vectorize a loop containing an 'unvectorizable' recurrence. Several of the compilers – notably those from Fujitsu, Hitachi, and NEC – were able to directly vectorize some of these loops.

We emphasize that 'beating the test' is not a bad thing. While there may be more efficient ways to vectorize the loops, the techniques above *are* beneficial.

8.3. Caveats

We caution that the results presented here test only one aspect of a compiler and should in no way be used to judge the overall performance of a vectorizing compiler or computer system. The results reflect only a limited spectrum of Fortran constructs. We do not claim these loops are representative of a 'real' workload, just that they make an interesting test. Some additional factors are discussed below.

8.3.1 Cache effects

Two issues may impact machines with data caches. First, to ensure a large enough granularity for timing purposes, we included a repetition loop around the loop of interest. While considered a necessary evil for test purposes, this artificial repetition raises an important question about data locality. The concern is that a cache machine will benefit from the reuse of data loaded into cache on the first trip through the repetition loop and that additional references to main memory will not be necessary.

The second issue concerns the data set size relative to the cache size. A small data set will always fit in the cache. A large data set may not fit in the cache and will cause many performance-degrading cache misses to occur. The paper by Abu-Sufah and Maloney [1] contains a discussion of this issue and its impact on performance. Their uniprocessor performance results on an Alliant FX/8 show that there is only a narrow range of vector lengths for which optimal performance was achieved. Our choice of 10, 100, and 1000 as the vector lengths was somewhat intuitive and was not made with any particular cache size in mind.

8.3.2 Loop granularity

Because of the small granularity of our loops (at most a few statements) the speedups achievable with a certain technique may not achievable on our particular loops. As an example, vectorizing the loop shown on the left in *Fig. 4* requires splitting the loop into two vectorizable loops and one scalar loop containing the nonlinear recurrence as shown on the right.

```
do i=2,n                            do vector i=2,n
   a(i)=a(i)+b(i)                      a(i)=a(i)+b(i)
   b(i)=b(i-1)*b(i-1)*a(i)          enddo
   a(i)=a(i)-b(i)                   do i=2,n
enddo                                  b(i)=b(i-1)*b(i-1)*a(i)
                                    enddo
                                    do vector i=2,n
                                       a(i)=a(i)-b(i)
                                    enddo
```

Fig. 4.

For this transformation to be successful, there needs to be enough work in the loop to justify the two additional loop overheads introduced and the extra loads and stores which are not required in the original loop. For this loop, inspection of the compiler listing showed that eight of the nine compilers had partially vectorized the loop, but only three achieved more than 15% of the estimated optimal performance, and only one achieved more than 50%.

8.3.3 Hardware and software

Some of the loops are really tests of the underlying hardware and may not accurately reflect the ability of the compiler itself. For example, in the statement A(I) = B(INDEX(I)) a compiler may detect the indirect addressing of array B but not generate vector instructions because the computer does not have hardware support for array references of this form. Other examples are loops containing IF tests that may require mask registers, or recurrences that require special library routines.

Several of the computers tested are multiprocessors whose compilers support the generation of both parallel and vector code (*Table 15*). Our test involved strictly uniprocessors and may have penalized vendors who have put considerable effort into parallel execution. On some of these machines, parallel execution may be more efficient than vectorization for certain loops.

Another example where the computer architecture may influence the compiler is on machines that have a data cache. Compilers for such machines may concentrate on loop transformations that improve data locality at the expense of adding 'simple' vectorization capabilities.

Several vendors have sophisticated tools to aid the user in vectorization. For example, both Fujitsu and NEC offer vectorization tools that interactively assist the user in vectorizing a program. Another example is an interprocedural analysis compiler from Convex, which analyzes an entire program at once. While all are very sophisticated tools, their use was against our rules.

9. Conclusion and future work

Our results indicate that most of the compilers tested are fairly sophisticated, able to use a wide variety of vectorization techniques and transformations. Loops that were considered challenging several years ago, such as indirect addressing or vectorizing loops containing multiple IF tests, now seem routine. While there are still various vectorization challenges left to be met, we are not sure how much they will be addressed in the future. Our perception is that most current compiler work is going into memory hiearchy management, parallel loop generation, highly pipelined scalar processors, and interactive and interprocedural tools. We may well be nearing a plateau as far as how much additional work vendors will put into vectorization techniques alone.

Our test suite continues to evolve from simple inspection of the compiler's output listing to trying to judge the quality of the execution results. To make the test more meaningful, we plan to add the types of 'real' loops found in applications. Real loops present combinations of vectorization problems rather than individual challenges. It will then be interesting to compare results on the 'simple' loops with those on the real loops.

A copy of the source code used in the test is available from the NETLIB electronic mail facility [4] at Oak Ridge National Laboratory. To receive a copy of the code, send electronic mail to *netlib@ornl.gov*. In the mail message, type *send vectors from benchmark* or *send vectord from benchmark* to get either the REAL or DOUBLE PRECISION versions, respectively.

Table 15
Hardware and software used in this test

Company	Compiler version	OS version
Computer	Compiler options	Main/cache memory
CONVEX Computer Corp.	fc 6.1	OS 9.0
CONVEX C210	-O2 -uo -is	512MB/None
Cray Computer Corp.	cft77 4.0.1.1	UNICOS 6.0
CRAY-2	Defaults	1GB/None
Cray Research, Inc.	CF77 4.0	UNICOS 5.1
CRAY Y-MP	-Wd"-e78b"	1GB/None
Digital Equipment Corp.	FORTRAN V5.5, HPO V1.0	VMS 5.4
VAXvector 9000-210	/HPO/VECTOR/BLAS = (INLINE,	512MB/128KB
	MAPPED)/ASSUME = NOACC/OPT	
FPS Computing	f77 4.3	FPX, 4.3.2
FPS M511EA-2	-u -O -Oc inl + -Oc vec + -Oc pi +	256MB/64KB
Fujitsu	Fortran77EX/VP V11L10	OSIV/MSP AFII &,
VP2600/10	VP(2600),OPT(F),INLINE(EXT(S151S))	VPCF V10L10
	VMSG(DETAIL)	1GB/None
Hitachi	fort77/hap V24-0f	vos3/as jss4 01-02
S-820/80	sopt,xfunc(xfr),	512MB/256KB
	hap(model80,vist),uinline	
IBM Corp.	VS FORTRAN 2.4.0, VAST-2 R2	MVS/ESA SP3.1.0E
IBM 3090-600J	vopt(opton = r8 inline = s151s,s152s)	JES2 SP3.1.1
	copt(opt(3) vec(rep(xlist)))	256MB/256KB
		512MB Extended Memory
NEC Corp.	f77sx 010	SUPER-UX R1.11
SX-X/14	-pi *:s151s *:s152s	1GB/64KB

Acknowledgements

We thank John Levesque, Murray Richman, Steve Wallach, Joel Williamson, and Michael Wolfe for providing many of the loops used in this test. Thanks also to the many people involved in running this test, providing results and constructive feedback on earlier versions.

References

[1] W. Abu-Sufah and A. Malony, Vector processing on the Alliant FX/8 multiprocessor, in: *Proc. 1986 Internat. Conf. on Parallel Processing* (1986) 559–566.

[2] J. Allen and K. Kennedy, Automatic translation of Fortran programs to vector form, *TOPLAS* 9(4) (1987) 491–542

[3] D. Callahan, J. Dongarra and D. Levine, Vectorizing compilers: A test suite and results, in: *Proc. Supercomputing '88*, (1988) 98–105.

[4] J. Dongarra and E. Grosse, Distribution of mathematical software via electronic mail, *Commun. ACM* 30(5) (Jul. 1987) 403–407.

[5] R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms* (Adam Hilger, Bristol, UK, 1981).

[6] J. Levesque and J. Williamson, *A Guidebook to Fortran on Supercomputers* (Academic Press, New York, 1988).

[7] D. Levine, D. Callahan and J. Dongarra, A comparative study of automatic vectorizing compilers, Technical Report MCS-P218-0391, Argonne National Laboratory, 1991.

[8] O. Lubeck, Supercomputer performance: The theory, practice, and results, Technical Report LA-11204-MS, Los Alamos National Laboratory, 1988.

[9] O. Lubeck, J. Moore and R. Mendez, A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2, *IEEE Comput.* 18(12) (1985) 10–23.

[10] W. Mangione-Smith, S. Abraham and E. Davidson, A performance comparison of the IBM RS/6000 and the Astronautics ZS-1, *IEEE Comput.* 24(1) (1991) 39–46.

[11] F. McMahon, The Livermore Fortran kernels: A computer test of the numercial range, Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[12] D. Padua and M. Wolfe, Advanced compiler optimizations for supercomputers, *Commun. ACM* 29(12) (1986) 1184–1201.

[13] J. Smith, Characterizing computing performance with a single number, *Commun. ACM* 31(10) (1988) 1202–1206.

[14] J. Tang and E. Davidson, An evaluation of Cray-1 and Cray X-MP performance on vectorizable Livermore Fortran kernels, in: *Proc. 1988 Internat. Conf. on Supercomputing*, St. Malo, France (1988) 510–518.

[15] M. Wolfe, *Optimizing Supercompilers for Supercomputers* (MIT Press, Cambridge, MA, 1989).