Vectorizing Compilers: A Test Suite and Results

David Callahan

Computer Science Department
Rice University
Houston, Texas 77251

Jack Dongarra* and David Levine*

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4801

ABSTRACT

This report describes a collection of 100 Fortran loops used to test the effectiveness of an automatic vectorizing compiler. We present the results of compiling these loops using commercially available, vectorizing Fortran compilers on a variety of supercomputers, mini-supercomputers, and mainframes.

1. Introduction

This report describes a collection of 100 Fortran loops used to test the effectiveness of an automatic vectorizing compiler. 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-8x array extensions or as subroutine calls to a vector library.

The loops in the test suite were written by people involved in the development of vectorizing compilers. Several of the loops 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 these loops using commercially available, vectorizing Fortran compilers on a variety of supercomputers, mini-supercomputers, and mainframes.

This paper is organized into seven sections. Section 2 categorizes the collection of loops used in the test. Section 3 describes the methodology used to perform the test. Section 4 explains how the results were scored. Section 5 presents the results of our testing, and Section 6 analyzes the results. Section 7 contains a discussion of the test suite. In Section 8 we make a few concluding remarks.

2. The Test Suite

The objective of the test suite is to test four broad areas of a vectorizing compiler: dependence analysis, vectorization, idiom recognition, and language completeness. For each of these areas, we have identified a number of subcategories. All of the loops in this test are classified into one of these categories**. A complete listing of the source code for the loops used in this test may be found in [3].

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

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 [1] and [4].

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 (e.g., the statement A(I)=A(I-1)+A(I) is vectorizable inside a DO I=1,N,2 loop but not a DO I=1,N,1 loop) 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 I=I+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

** Not all the subcategories listed are represented in this report. Missing are subcategories 1.4, 2.6, 2.10, 4.3, 4.9, and 4.12. Over time we plan to add tests for these categories to complete the current set.

(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)).

2.3 Idiom Recognition

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

1. *Reductions.* Computation of a scalar value or values from a vector, such as sum reductions, min/max reductions, dot products, and product reductions.
2. *Recurrences.* Special first- and second-order recurrences that have logarithmically faster solutions or hardware support.
3. *Search Loops.* Searching 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.

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.* Recognition and vectorization of loops formed by backward GO TO's.
2. *Storage Classes and Equivalencing.* Understanding of the scope of local vs. common storage; correct handling of equivalencing.
3. *Parameters.* Analysis of symbolic named constants and vectorization of statements that refer to them.
4. *Non-logical IF's.* Vectorization of loops containing computed GO TO's, assigned GO TO's, arithmetic GO TO's, alternative returns from CALL statements, and END= clauses on I/O statements.
5. *Intrinsic Functions.* Vectorization of or around functions that have elemental (vector) versions such as SIN and COS or known side effects.
6. *I/O Statements.* Vectorization of statements in loops that contain I/O statements.
7. *Call Statements.* Vectorization of statements in loops that contain CALL statements or external function invocations.
8. *Non-local GO TO's.* Branches out of loops and RETURN or STOP statements inside loops.
9. *Vector Semantics.* Load before store and preservation of order of stores.
10. *Data Types.* Vectorization of COMPLEX and INTEGER as well as REAL.
11. *Indirect Addressing.* Vectorization of subscripted subscript references (e.g, A(INDEX(I))) as Gather/Scatter.
12. *Statement Functions.* Vectorization of statements that refer to Fortran statement functions.

3. Testing Methodology

Vendors were mailed a magnetic tape containing all the loops we had collected. They were asked to compile the loops without making any changes* using only the compiler options for automatic vectorization. Thus, the use of compiler directives or interactive compilation features to gain additional vectorizations was not tested. Further, many runtime details of vectorization and what Arnold [2] and Wolfe [5] refer to as "vector optimization" are not tested.

After compiling the loops, the vendors sent back the compiler's output listing (source echo, diagnostics, and messages). We then examined these listings to see which loops had been vectorized. No attempt was made to execute the loops to verify the

* Separate compilation of the subroutines used for interprocedural analysis testing was permitted.

correctness of the compiler-generated code or to measure the efficiency of the code when run.

We mailed a total of approximately 240 loops to each vendor. These consisted of all the different loops we had collected over the past several years. From this set we selected the 100 whose results are presented in this paper. This was done by eliminating loops that tested the same or similar features, tested vector optimization, or contained errors of some sort. The selection of which loops to include was made previous to, and independent of, receiving the results from the vendors.

4. Loop Scoring

We define a statement as vectorizable if one or more of the expressions in the statement involve array references or may be converted to that form. All loops in the test suite consist of one or more such statements.

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 a few loops the IBM and Amdahl compilers generated scalar code even though the compiler indicated vector code was possible. This was because for those loops, scalar code was more efficient for their machines. These loops have been scored as *vectorized* and *partially vectorized*, as appropriate.

The Cray CF77 and CFT77 compiler's conditionally vectorized certain loops. This means that for loops with ambiguous subscripts, a runtime test was compiled that selected a safe vector length†. These loops have been scored as either *vectorized* or *not vectorized* according to whether or not vectorized code would actually be executed at runtime.

For some loops the Cray CFT compiler 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 would actually be executed at runtime.

The Alliant, Ardent, Convex, and Stellar compilers support the generation of both parallel and vector code. For some loops the Alliant, Ardent, and Convex compilers generated parallel code but not vector code. This may be because the loop was difficult to vectorize but simple to parallelize, or because parallel execution was the most efficient on these machines. These loops have been scored as *not vectorized*.

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. In this and similar cases, the loop is still scored as *not vectorized*.

† A safe vector length is one which allows the compiler to execute vector instructions and still produce the correct result. E.g., 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.

5. Results

Tables 1-6 list the results of compiling this set of loops on different computers. Table 1 summarizes the results for all 100 loops. Table 2 is also a summary of all the loops; here, however, the column P/V gives a count of loops that were either fully or partially vectorized. Tables 3-6 contain results by category as defined in Section 2.

Table 1. Summary of Test Suite (100 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V4.0	68	5	27
Amdahl VP-E Series	Fortran 77/VP V10L30	62	11	27
Ardent Titan-1	Fortran V1.0	62	6	32
CDC Cyber 205	VAST-2 V2.21	62	5	33
CDC Cyber 990E/995E	VFTN V2.1	25	11	64
Convex C Series	FC5.0	69	5	26
Cray Series	CF77 V3.0	69	3	28
CRAY X-MP	CFT V1.15	50	1	49
Cray Series	CFT77 V3.0	50	1	49
CRAY-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Gould NP1	GCF 2.0	60	7	33
Hitachi S-810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS Fortran V2.4	52	4	44
Intel iPSC/2-VX	VAST-2 V2.23	56	8	36
NEC SX/2	FORTRAN77/SX V.040	66	5	29
SCS-40	CFT x13g	24	1	75
Stellar GS 1000	F77 prerelease	48	11	41
Unisys ISP	UFTN 4.1.2	67	13	20

Key to symbols for Tables 1-6

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

Table 2. Full and Partial Vectorization (100 loops)

Machine	Compiler	V/P	N
Alliant FX/8	FX/Fortran V4.0	73	27
Amdahl VP-E Series	Fortran 77/VP V10L30	73	27
Ardent Titan-1	Fortran V1.0	68	32
CDC Cyber 205	VAST-2 V2.21	67	33
CDC Cyber 990E/995E	VFTN V2.1	36	64
Convex C Series	FC5.0	74	26
Cray Series	CF77 V3.0	72	28
CRAY X-MP	CFT V1.15	51	49
Cray Series	CFT77 V3.0	51	49
CRAY-2	CFT2 V3.1a	28	72
ETA-10	FTN 77 V1.0	69	31
Gould NP1	GCF 2.0	67	33
Hitachi S-810/820	FORT77/HAP V20-2B	71	29
IBM 3090/VF	VS Fortran V2.4	56	44
Intel iPSC/2-VX	VAST-2 V2.23	64	36
NEC SX/2	FORTRAN77/SX V.040	71	29
SCS-40	CFT x13g	25	75
Stellar GS 1000	F77 prerelease	59	41
Unisys ISP	UFTN 4.1.2	80	20

Table 3. Dependence Analysis (24 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V4.0	19	0	5
Amdahl VP-E Series	Fortran 77/VP V10L30	16	1	7
Ardent Titan-1	Fortran V1.0	18	0	6
CDC Cyber 205	VAST-2 V2.21	16	0	8
CDC Cyber 990E/995E	VFTN V2.1	8	0	16
Convex C Series	FC5.0	17	0	7
Cray Series	CF77 V3.0	20	0	4
CRAY X-MP	CFT V1.15	16	0	8
Cray Series	CFT77 V3.0	17	0	7
CRAY-2	CFT2 V3.1a	5	0	19
ETA-10	FTN 77 V1.0	18	0	6
Gould NP1	GCF 2.0	14	0	10
Hitachi S-810/820	FORT77/HAP V20-2B	14	0	10
IBM 3090/VF	VS Fortran V2.4	12	0	12
Intel iPSC/2-VX	VAST-2 V2.23	15	0	9
NEC SX/2	FORTRAN77/SX V.040	17	0	7
SCS-40	CFT x13g	7	0	17
Stellar GS 1000	F77 prerelease	14	0	10
Unisys ISP	UFTN 4.1.2	21	3	0

Table 4. Vectorization (34 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V4.0	20	5	9
Amdahl VP-E Series	Fortran 77/VP V10L30	21	8	5
Ardent Titan-1	Fortran V1.0	19	5	10
CDC Cyber 205	VAST-2 V2.21	20	5	9
CDC Cyber 990E/995E	VFTN V2.1	6	8	20
Convex C Series	FC5.0	25	4	5
Cray Series	CF77 V3.0	18	3	13
CRAY X-MP	CFT V1.15	12	1	21
Cray Series	CFT77 V3.0	8	1	25
CRAY-2	CFT2 V3.1a	3	1	30
ETA-10	FTN 77 V1.0	18	7	9
Gould NP1	GCF 2.0	19	7	8
Hitachi S-810/820	FORT77/HAP V20-2B	24	4	6
IBM 3090/VF	VS Fortran V2.4	19	3	12
Intel iPSC/2-VX	VAST-2 V2.23	17	8	9
NEC SX/2	FORTRAN77/SX V.040	21	5	8
SCS-40	CFT x13g	6	1	27
Stellar GS 1000	F77 prerelease	20	9	5
Unisys ISP	UFTN 4.1.2	19	8	7

Table 5. Idiom Recognition (15 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V4.0	10	0	5
Amdahl VP-E Series	Fortran 77/VP V10L30	11	1	3
Ardent Titan-1	Fortran V1.0	9	0	6
CDC Cyber 205	VAST-2 V2.21	7	0	8
CDC Cyber 990E/995E	VFTN V2.1	3	1	11
Convex C Series	FC5.0	11	0	4
Cray Series	CF77 V3.0	9	0	6
CRAY X-MP	CFT V1.15	10	0	5
Cray Series	CFT77 V3.0	7	0	8
CRAY-2	CFT2 V3.1a	8	0	7
ETA-10	FTN 77 V1.0	7	0	8
Gould NP1	GCF 2.0	8	0	7
Hitachi S-810/820	FORT77/HAP V20-2B	14	0	1
IBM 3090/VF	VS Fortran V2.4	5	1	9
Intel iPSC/2-VX	VAST-2 V2.23	6	0	9
NEC SX/2	FORTRAN77/SX V.040	12	0	3
SCS-40	CFT x13g	5	0	10
Stellar GS 1000	F77 prerelease	4	1	10
Unisys ISP	UFTN 4.1.2	10	2	3

Table 6. Language Completeness (27 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V4.0	19	0	8
Amdahl VP-E Series	Fortran 77/VP V10L30	14	1	12
Ardent Titan-1	Fortran V1.0	16	1	10
CDC Cyber 205	VAST-2 V2.21	19	0	8
CDC Cyber 990E/995E	VFTN V2.1	8	2	17
Convex C Series	FC5.0	16	1	10
Cray Series	CF77 V3.0	22	0	5
CRAY X-MP	CFT V1.15	12	0	15
Cray Series	CFT77 V3.0	18	0	9
CRAY-2	CFT2 V3.1a	11	0	16
ETA-10	FTN 77 V1.0	19	0	8
Gould NP1	GCF 2.0	19	0	8
Hitachi S-810/820	FORT77/HAP V20-2B	15	0	12
IBM 3090/VF	VS Fortran V2.4	16	0	11
Intel iPSC/2-VX	VAST-2 V2.23	18	0	9
NEC SX/2	FORTRAN77/SX V.040	16	0	11
SCS-40	CFT x13g	6	0	21
Stellar GS 1000	F77 prerelease	10	1	16
Unisys ISP	UFTN 4.1.2	17	0	10

6. Analysis of Results

The average number of loops vectorized (Table 1) was 55%, and vectorized or partially vectorized (Table 2) was 61%. The best results were 69% and 80%, respectively. Of the 100 loops, only 4 were not vectorized or partially vectorized by any of the compilers. All 4 loops can be vectorized by a knowledgeable programmer. There is probably no significant difference between vendors 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.

Comparing Table 1 to Table 2, we see that the inclusion of partially vectorized loops in the totals places the Amdahl and Unisys compilers among the top performers. Similarly the CDC Cyber 990E/995E and Stellar compilers, which also did a significant amount of partial vectorization, moved up in the list.

Tables 3-6 show that some compilers did particularly well in certain categories. In the Dependence Analysis category, Unisys vectorized or partially vectorized all 24 loops. Convex had the best result in the Vectorization category, vectorizing 25 and partially vectorizing 4 of the 34 loops. Hitachi did very well in the section on Idiom Recognition, vectorizing 14 of the 15 loops. Cray's CF77 compiler had the best result in the Language Completeness category, vectorizing 22 of the 27 loops.

In analyzing the results we found that some vendors, with approximately equal results, did much better in one category than another. Interprocedural analysis, recognizing loops formed by IF and GOTO statements, and vectorizing loops containing COMMON or EQUIVALENCE statements are examples of such categories. 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.

The results reported in this paper were collected over a period of approximately a year. Many of the results are from compilers in production use. Some compilers were in beta test (Alliant FX/Fortran V4.0, Convex FC V5.0, Cray CFT77 V3.0, NEC FORTRAN77/SX V.040), some were in various stages of development of the next release (Gould V2.0, IBM V2.4, Unisys V4.1.2), and some were prerelease systems (Stellar F77, Cray CF77).

In the following subsections, we discuss the results in more detail using the categories defined in Section 2. Results, on a loop-by-loop basis, for all vendors may be found in [3].

6.1 Dependence Analysis

Most compilers vectorized a majority of the linear dependence tests, which analyze subscript expressions and loop control variables to detect access to the same memory location. In the induction variable recognition tests the Unisys compiler did the best job; it vectorized or partially vectorized all the loops. Ardent, Cray CF77, Hitachi and the VAST systems* also did a good job, missing only the loops with an induction variable under an IF.

Alliant, Amdahl, Ardent, Convex, Cray CF77, ETA, and Unisys vectorized both interprocedural analysis tests, and the CFT77 compiler vectorized one, all via a procedure integration capability. The Cray CFT compiler also vectorized one of these loops using a runtime test.

Most compilers did very well in the symbolics section, where the information necessary to recognize dependencies is contained in the loop bounds. Many were also able to do the global analysis necessary to vectorize statements like A(I) = A(I+M) where the value of M was supplied outside the loop block.

6.2 Vectorization

The Vectorization category contained the largest amount of partial vectorization. Most compilers were able to vectorize the tests that required reordering statements within a loop. In loop distribution testing we found most compilers able to do partial vectorization; Alliant, Amdahl, CDC Cyber 205, Hitachi, NEC, and Unisys with their capabilities for vectorizing recurrences completely vectorized at least one of the loops.

Loop interchange was a challenging section: all vendors missed at least 2 of the 4 loops. There were a variety of results in the node-splitting tests. Some vendors, particularly those that could vectorize recurrences, did quite well vectorizing or partially vectorizing most or all of the loops. The scalar expansion tests also showed varied results. All vendors were able to vectorize at least one of the loops. Ardent, Convex, and Hitachi vectorized all 5 loops.

Many vendors did very well in vectorizing loops containing IF tests. Unisys vectorized all 12 loops. Amdahl, Convex, IBM, and Stellar each vectorized 11. Cray CF77, Hitachi, NEC, and the VAST systems vectorized 10.

The tests for crossing thresholds and loop peeling were among the most difficult. These tests require breaking up the loop or peeling off some iterations. Hitachi and NEC were the most successful, followed by Ardent and Convex.

* We use the term VAST systems to refer to the Alliant FX/8, CDC Cyber 205, ETA-10, Gould NP1, and Intel iPSC-VX compilers, all of which were using Pacific Sierra's VAST product as a front-end.

6.3 Idiom Recognition

Idiom recognition, more than the other categories, relies on special-purpose hardware or software to enable the compiler to vectorize some of the loops. All systems vectorized sum and dot product reductions. Most also vectorized product reduction, loops to find the maximum or minimum element in an array, and an unrolled dot product loop. First-order recurrences were vectorized by Alliant, Amdahl, Cray CFT2, Hitachi, NEC, and Unisys. Only Alliant vectorized a second-order recurrence. Only Hitachi vectorized a coupled recurrence.

Many systems had trouble vectorizing search loops and loops that packed or unpacked an array. Both types of loops require an element-by-element search through an array, under the control of an IF test, to look for a certain condition. Amdahl and Hitachi vectorized all 4 loops. Convex, Cray CFT, and NEC vectorized 3 of the loops.

6.4 Language Completeness

Cray CF77 and the VAST systems were the only compilers to vectorize loops formed by IF and GOTO statements. Vectorization of loops containing COMMON and EQUIVALENCE statements showed interesting results. All vendors vectorized either most (5-7) of the 7 loops, or else just 1 or 2. CDC Cyber 205, Cray CF77, and Cray CFT77 vectorized all 7 loops.

There were mixed results in vectorizing loops containing various Fortran constructs. Some of the difficult loops that were vectorized contained an arithmetic IF statement, a WRITE statement, a CALL statement, and a STOP statement. Doing well in these tests were Amdahl, Convex, Cray CF77, CDC Cyber 990E/995E, and Hitachi. Most systems vectorized a loop containing the SIN and COS intrinsic functions.

Almost all of the compilers vectorized Gather/Scatter loops. We believe those compilers that did not vectorize these loops currently lack the hardware necessary to support this type of addressing.

7. Discussion

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.

7.1 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 a few 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 published literature on 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 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 (34 loops) than on idiom recognition (15 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.

7.2 Stress

By "stress" we refer to how effectively the test suite tests the limits of the compilers. We want 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 7 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 7 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. Only in a few cases were loops vectorized by some vendors and only partially vectorized by others. We interpret a low score as an indication of a difficult test. From the table we observe a good distribution of test difficulties from "easy" (everyone vectorizes) to "difficult" (no one even partially vectorizes).

This method of judging difficulty will be skewed if many of the compilers are similar. Using the performance of the compilers to measure the difficulty of the loops assumes that each compiler is an independent measure. When there are significant relationships between compilers, loops may seem artificially easy or difficult depending on whether the related compilers all vectorize or all fail to vectorize. As an example, in our suite, 6 of the 19 compilers vectorize implicit loops constructed from backward GOTO's. Five of these are based on Pacific Sierra Research's's VAST system. The effect on the scoring is that these loops seem easier than some others. On the other hand, in a few cases the VAST systems did comparatively poorly on some loops. Here, the effect on the scoring is that these loops appear more difficult than is perhaps true. Similar relationships exist among some of the other compilers reported on in this paper.

Table 8 contains a matrix of linear correlations between the performance of the systems. Since a large number of loops were vectorized by most of the systems, all of the correlations shown in Table 8 are positive. The correlations between Alliant, CDC Cyber 205, ETA, Gould, and Intel (the VAST systems) are all between 0.81 and 0.91, significantly higher than all other correlations. If we ignore the easy and the difficult tests, more variation appears. Table 9 is the correlation matrix restricted to tests that were vectorized or

Table 7. Loops Sorted by Difficulty

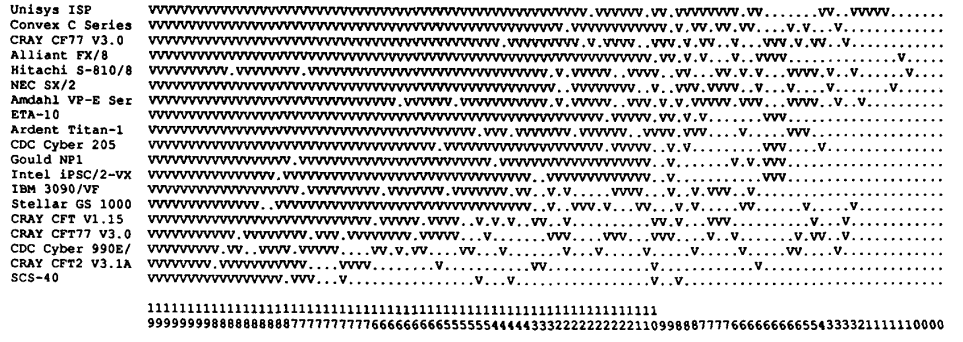

Table 8. Correlation of Compiler Performance

Unisys ISP	1.00																	
Convex C Series	0.50	1.00																
Alliant FX/8	0.43	0.51	1.00															
Amdahl VP-E	0.37	0.72	0.34	1.00														
CRAY CF77 V3.0	0.36	0.34	0.67	0.22	1.00													
Hitachi S-810/8	0.34	0.58	0.36	0.60	0.24	1.00												
NEC SX/2	0.62	0.63	0.50	0.46	0.44	0.56	1.00											
ETA-10	0.42	0.59	0.91	0.37	0.69	0.38	0.48	1.00										
Ardent Titan-1	0.35	0.67	0.60	0.45	0.53	0.46	0.51	0.65	1.00									
CDC Cyber 205	0.39	0.51	0.82	0.29	0.65	0.40	0.49	0.91	0.57	1.00								
Gould NP1	0.39	0.55	0.82	0.34	0.60	0.35	0.49	0.82	0.52	0.82	1.00							
Intel iPSC/2-VX	0.35	0.55	0.81	0.34	0.60	0.39	0.48	0.85	0.56	0.85	0.85	1.00						
Stellar GS 1000	0.40	0.66	0.45	0.55	0.30	0.54	0.50	0.50	0.52	0.50	0.54	0.60	1.00					
IBM 3090/VF	0.51	0.53	0.46	0.37	0.43	0.32	0.54	0.54	0.47	0.53	0.53	0.59	0.53	1.00				
CRAY CFT V1.15	0.41	0.42	0.31	0.40	0.50	0.39	0.48	0.38	0.36	0.33	0.29	0.31	0.36	0.46	1.00			
CRAY CFT77 V3.0	0.31	0.24	0.35	0.12	0.64	0.12	0.30	0.34	0.49	0.33	0.38	0.39	0.24	0.42	0.52	1.00		
CDC Cyber 990E	0.11	0.35	0.22	0.36	0.14	0.25	0.25	0.28	0.47	0.30	0.30	0.30	0.33	0.33	0.24	0.24	1.00	
CRAY CFT2 V3.1A	0.31	0.32	0.33	0.33	0.34	0.30	0.40	0.32	0.38	0.30	0.30	0.24	0.25	0.33	0.57	0.39	0.23	1.00
SCS-40	0.29	0.34	0.25	0.30	0.31	0.27	0.37	0.29	0.25	0.31	0.26	0.29	0.34	0.33	0.57	0.29	0.34	0.67

Table 9. Correlation of Selected Compiler Performance

Unisys ISP	1.00																	
Convex C Series	0.12	1.00																
Alliant FX/8	0.30	-0.02	1.00															
Amdahl VP-E	-0.14	0.44	-0.42	1.00														
CRAY CF77 V3.0	0.05	-0.43	0.42	-0.49	1.00													
Hitachi S-810/8	-0.31	0.19	-0.14	0.52	-0.39	1.00												
NEC SX/2	0.41	0.09	0.04	-0.10	-0.08	0.11	1.00											
ETA-10	0.20	0.20	0.86	-0.31	0.43	-0.10	-0.08	1.00										
Ardent Titan-1	-0.35	0.14	0.25	-0.24	0.13	-0.14	-0.25	0.42	1.00									
CDC Cyber 205	0.11	0.11	0.75	-0.39	0.29	0.10	0.08	0.87	0.31	1.00								
Gould NP1	0.16	0.16	0.66	-0.35	0.21	-0.15	0.00	0.64	0.07	0.79	1.00							
Intel iPSC/2-VX	0.07	0.23	0.70	-0.25	0.23	0.04	0.02	0.81	0.25	0.93	0.87	1.00						
Stellar GS 1000	-0.19	0.47	-0.16	0.39	-0.50	0.24	-0.11	-0.05	-0.01	0.05	0.15	0.11	1.00					
IBM 3090/VF	0.35	0.02	-0.10	-0.29	0.02	-0.47	0.11	0.02	-0.25	0.13	0.22	0.19	-0.14	1.00				
CRAY CFT V1.15	0.18	-0.18	-0.38	0.10	0.04	-0.09	0.00	-0.29	-0.38	-0.37	-0.41	-0.33	-0.26	0.21	1.00			
CRAY CFT77 V3.0	-0.02	-0.52	0.14	-0.52	0.69	-0.53	-0.26	0.10	0.14	0.05	0.15	0.11	-0.56	0.16	0.26	1.00		
CDC Cyber 990E	-0.37	0.00	-0.21	0.10	-0.29	0.09	-0.17	-0.12	0.46	-0.04	-0.08	0.00	0.09	-0.13	-0.14	-0.09	1.00	
CRAY CFT2 V3.1A	0.12	-0.30	0.15	0.10	-0.21	0.14	0.17	-0.21	-0.24	-0.18	-0.19	-0.17	-0.14	-0.15	-0.11	-0.14	-0.11	1.00
SCS-40	0.12	0.12	-0.24	0.10	-0.21	0.14	0.17	-0.21	-0.24	-0.18	-0.19	-0.17	0.26	-0.15	0.33	-0.14	-0.11	-0.04

partially vectorized by no more than half the highest score nor less than one fourth of the maximum score. Scores were computed with only one VAST system represented. This table still shows high positive correlations between the VAST systems but also shows some high negative correlations, such as -0.49 between Cray CF77 V3.0 and Amdahl. We assume these negative correlations are the effects of different machine architectures and different decisions by the vendors about what is important or worthwhile.

7.3 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. The negative correlations in Table 9 support the view that multiple distinct, but correlated factors are involved.

We conclude that the suite represents reasonable coverage and adequate stress, but that we cannot determine the accuracy of the suite.

8. Conclusion

Our initial goal has been twofold: (1) to compare the ability of different Fortran compilers to automatically vectorize various loops, and (2) to try to understand their capabilities and limitations. The real test of a vectorizing compiler can be determined only by actually comparing the execution time of the vectorized and non-vectorized code. We caution that the information presented here tests 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. Also, subsequent compiler and hardware changes may affect which loops can be vectorized.

We intend to update and expand the results presented here. In particular, we plan to develop a check to verify the correctness of the compiler-generated code and a measure of its efficiency. A copy of the source code used in the test is available from netlib at Argonne National Laboratory. To receive a copy of the code, send electronic mail to netlib@anl-mcs.arpa. In the mail message, type:

send vector from benchmark

Acknowledgments

We thank all the people who have helped us put together this collection of loops and results. Particular thanks are given to people at IBM in Kingston, N.Y., Steve Wallach of Convex Computer Corporation, and Michael Wolfe of Kuck & Associates who provided an initial set of loops used as the basis of this test.

REFERENCES

[1] J. R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *TOPLAS*, vol. 9, no. 4, pp. 491-542, October 1987.

[2] C. Arnold, "Vector optimization on the Cyber 205," Proc. of the International Conf. for Parallel Processing, pp. 530-536, August 1983.

[3] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," Argonne National Laboratory MCS-TM-109, March 1988.

[4] D. Padua and M. Wolfe, "Advanced compiler optimizations for supercomputers," *CACM*, vol. 29, no. 12, pp. 1184-1201, December 1986.

[5] M. Wolfe, "Vector optimization vs. vectorization," Proc. of the 1987 Conf. on Supercomputing, Athens, Greece, June 1987.