ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Vectorizing Compilers: A Test Suite and Results

David Callahan, Jack Dongarra, and David Levine

Mathematics and Computer Science Division

Technical Memorandum No. 109

March 31, 1988

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 60439-4844

March 31, 1988

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.

Most of 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. In Section 7 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

* 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.

categories*. Appendix B contains a listing of the source code for the loops used in this test.

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 [1] and Padua and Wolfe [3] .

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 (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.

* Not all the subcategories listed are represented in this report. Over time we plan to add tests for these categories to complete the current set.

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 [4] 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 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, and are footnoted in Appendix A.

The Cray CFT77 compiler 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. They are marked with a footnote in Appendix A.

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. They are marked with a footnote in Appendix A.

---

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

† 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.

The Alliant, Ardent, and Stellar compilers support the generation of both parallel and vector code. For some loops the Alliant and Ardent 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* and are footnoted in Appendix A.

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.*

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 V3.0.14	65	6	29
Amdahl 1200/1400	Fortran 77/VP V10L20	59	12	29
Ardent Titan-1	Fortran prerelease	51	7	42
CDC Cyber 205	VAST-2 V2.21	62	5	33
CDC Cyber 990E/995E	VFTN V2.1	25	11	64
Convex C series	FC4.0	67	5	28
CRAY X-MP	CFT V1.15	50	1	49
CRAY X-MP	CFT77 V2.0	51	1	48
CRAY-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Gould NP1	VAST-2 prerelease	46	5	49
Hitachi S-810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS Fortran V2.3.0	38	4	58
Intel iPSC-VX	VAST-2 V2.22	56	8	36
NEC SX/2	FORTRAN77/SX	54	5	41
SCS-40	CFT x13g	24	1	75
Stellar GS 1000	F77 prerelease	48	11	41
Unisys ISP	UFTN 3.2.14	58	12	30

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 V3.0.14	71	29
Amdahl 1200/1400	Fortran 77/VP V10L20	71	29
Ardent Titan-1	Fortran prerelease	58	42
CDC Cyber 205	VAST-2 V2.21	67	33
CDC Cyber 990E/995E	VFTN V2.1	36	64
Convex C series	FC4.0	72	28
CRAY X-MP	CFT V1.15	51	49
CRAY X-MP	CFT77 V2.0	52	48
CRAY-2	CFT2 V3.1a	28	72
ETA-10	FTN 77 V1.0	69	31
Gould NP1	VAST-2 prerelease	51	49
Hitachi S-810/820	FORT77/HAP V20-2B	71	29
IBM 3090/VF	VS Fortran V2.3.0	42	58
Intel iPSC-VX	VAST-2 V2.22	64	36
NEC SX/2	FORTRAN77/SX	59	41
SCS-40	CFT x13g	25	75
Stellar GS 1000	F77 prerelease	59	41
Unisys ISP	UFTN 3.2.14	70	30

Table 3. Dependence Analysis (24 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V3.0.14	17	0	7
Amdahl 1200/1400	Fortran 77/VP V10L20	16	0	8
Ardent Titan-1	Fortran prerelease	14	0	10
CDC Cyber 205	VAST-2 V2.21	16	0	8
CDC Cyber 990E/995E	VFTN V2.1	8	0	16
Convex C series	FC4.0	17	0	7
CRAY X-MP	CFT V1.15	16	0	8
CRAY X-MP	CFT77 V2.0	17	0	7
CRAY-2	CFT2 V3.1a	5	0	19
ETA-10	FTN 77 V1.0	18	0	6
Gould NP1	VAST-2 prerelease	8	0	16
Hitachi S-810/820	FORT77/HAP V20-2B	14	0	10
IBM 3090/VF	VS Fortran V2.3.0	8	0	16
Intel iPSC-VX	VAST-2 V2.22	15	0	9
NEC SX/2	FORTRAN77/SX	14	0	10
SCS-40	CFT x13g	7	0	17
Stellar GS 1000	F77 prerelease	14	0	10
Unisys ISP	UFTN 3.2.14	20	3	1

Table 4. Vectorization (34 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V3.0.14	19	6	9
Amdahl 1200/1400	Fortran 77/VP V10L20	19	10	5
Ardent Titan-1	Fortran prerelease	17	5	12
CDC Cyber 205	VAST-2 V2.21	20	5	9
CDC Cyber 990E/995E	VFTN V2.1	6	8	20
Convex C series	FC4.0	25	4	5
CRAY X-MP	CFT V1.15	12	1	21
CRAY X-MP	CFT77 V2.0	9	1	24
CRAY-2	CFT2 V3.1a	3	1	30
ETA-10	FTN 77 V1.0	18	7	9
Gould NP1	VAST-2 prerelease	19	5	10
Hitachi S-810/820	FORT77/HAP V20-2B	24	4	6
IBM 3090/VF	VS Fortran V2.3.0	19	3	12
Intel iPSC-VX	VAST-2 V2.22	17	8	9
NEC SX/2	FORTRAN77/SX	20	5	9
SCS-40	CFT x13g	6	1	27
Stellar GS 1000	F77 prerelease	20	9	5
Unisys ISP	UFTN 3.2.14	17	7	10

Table 5. Idiom Recognition (15 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V3.0.14	10	0	5
Amdahl 1200/1400	Fortran 77/VP V10L20	10	1	4
Ardent Titan-1	Fortran prerelease	7	0	8
CDC Cyber 205	VAST-2 V2.21	7	0	8
CDC Cyber 990E/995E	VFTN V2.1	3	1	11
Convex C series	FC4.0	9	0	6
CRAY X-MP	CFT V1.15	10	0	5
CRAY X-MP	CFT77 V2.0	7	0	8
CRAY-2	CFT2 V3.1a	8	0	7
ETA-10	FTN 77 V1.0	7	0	8
Gould NP1	VAST-2 prerelease	8	0	7
Hitachi S-810/820	FORT77/HAP V20-2B	14	0	1
IBM 3090/VF	VS Fortran V2.3.0	3	1	11
Intel iPSC-VX	VAST-2 V2.22	6	0	9
NEC SX/2	FORTRAN77/SX	9	0	6
SCS-40	CFT x13g	5	0	10
Stellar GS 1000	F77 prerelease	4	1	10
Unisys ISP	UFTN 3.2.14	10	2	3

Table 6. Language Completeness (27 loops)

Machine	Compiler	V	P	N
Alliant FX/8	FX/Fortran V3.0.14	19	0	8
Amdahl 1200/1400	Fortran 77/VP V10L20	14	1	12
Ardent Titan-1	Fortran prerelease	13	2	12
CDC Cyber 205	VAST-2 V2.21	19	0	8
CDC Cyber 990E/995E	VFTN V2.1	8	2	17
Convex C series	FC4.0	16	1	10
CRAY X-MP	CFT V1.15	12	0	15
CRAY X-MP	CFT77 V2.0	18	0	9
CRAY-2	CFT2 V3.1a	11	0	16
ETA-10	FTN 77 V1.0	19	0	8
Gould NP1	VAST-2 prerelease	11	0	16
Hitachi S-810/820	FORT77/HAP V20-2B	15	0	12
IBM 3090/VF	VS Fortran V2.3.0	8	0	19
Intel iPSC-VX	VAST-2 V2.22	18	0	9
NEC SX/2	FORTRAN77/SX	11	0	16
SCS-40	CFT x13g	6	0	21
Stellar GS 1000	F77 prerelease	10	1	16
Unisys ISP	UFTN 3.2.14	11	0	16

6. Analysis of Results

The average number of loops vectorized or partially vectorized (Table 2) was 56%. The best result was 72%. 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 23 of the 24 loops. Hitachi did very well in the section on Idiom Recognition, vectorizing 14 of the 15 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 tests for the Ardent, Gould, and Stellar machines were all made on prerelease systems. We expect hardware and software developments to increase the number of loops these systems vectorize.

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 compilers may be found in Appendix A.

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. Hitachi and the VAST systems* also did a good job, missing only the loops with an induction variable under an IF.

Amdahl, Ardent, Convex, and ETA were the only vendors to vectorize both interprocedural analysis tests, all via a procedure integration capability. The Unisys and Cray CFT and CFT77 compilers also vectorized one of these loops using a runtime test.

Only a few compilers appeared 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. However, most compilers did well in the symbolics section, where the information necessary to recognize dependencies is contained in the loop bounds.

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, CDC Cyber 205, Hitachi, and NEC with their capabilities for vectorizing recurrences completely vectorized at least one of the loops.

Loop interchange was a challenging section: all vendors missed some of the loops. There were a variety of results in the node-splitting tests, with some vendors doing fairly well and others missing all the loops. Most compilers did a good job with the loops that test for scalar expansion, although two of the loops that dealt with carryaround variables were either missed or only partially vectorized by many compilers.

Many vendors did very well in vectorizing loops containing IF tests. Unisys vectorized all 12 loops. Amdahl, Convex, IBM, and Stellar each vectorized 11. 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 was the most successful, followed by Ardent and Convex, and to a lesser extent Amdahl, NEC, and Stellar.

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, Cray CFT2, Hitachi, NEC, and Unisys. Only Alliant vectorized the second-order 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 of these loops. Convex, Unisys, and the Cray CFT and CFT77 compilers vectorized or partially vectorized most of the loops.

6.4. Language Completeness

Only the VAST systems vectorized loops formed by IF and GOTO statements. There was considerable variation in vectorizing loops containing COMMON and EQUIVALENCE statements. Ardent, Cray CFT77, and the VAST systems did very well; others were unable to vectorize any of these loops.

---

* 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.

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, and CDC Cyber 990E/995E. 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. Conclusion

Our initial goal has been to compare the ability of different Fortran compilers to automatically vectorize various loops. 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. Padua and M. Wolfe, ‘‘Advanced Compiler Optimizations for Supercomputers,’’ *CACM*, vol. 29, no. 12, pp. 1184-1201, December 1986.
4. M. Wolfe, ‘‘Vector Optimization vs. Vectorization,’’ Proc. of the 1987 Conf. on Supercomputing, Athens, Greece, June 1987.

Appendix A

RESULTS BY LOOP FOR ALL COMPILERS

	s111	s112	s113	s114	s115	s116	s117	s121
Alliant FX/8	v	v	v	v	v	n	v	v
Amdahl 1200/1400	v	v	v	v	v	v	n	v
Ardent Titan-1	v	v	v	n	n	n	n	v
CDC CYBER 205	v	v	v	v	v	n	n	v
CDC CYBER 990	v	v	v	n	v	n	n	n
Convex C series	v	v	v	v	v	v	n	v
CRAY X-MP CFT	v	v	n	v	v [a]	v	n	v
CRAY X-MP CFT77	v	v	v	n	v [b]	n	v [b]	v
CRAY-2	n	v	n	n	n	n	n	n
ETA-10	v	v	v	v	v	n	n	v
Gould NP1	n	n	v	n	v	n	v	v
Hitachi S-810/820	v	v	v	v	v	v	n	v
IBM 3090/VF	v	v	v	n	n	n	n	v
Intel iPSC-VX	v	v	v	v	v	n	n	v
NEC SX/2	v	v	v	v	v	v	n	v
SCS-40	v	v	n	n	n	v	n	n
Stellar GS1000	v	v	v	v	v	v	n	v
Unisys ISP	v	v	v	v	v	v	p	v

	s122	s123	s124	s125	s126	s127	s131	s132
Alliant FX/8	v	n	n	v	v	v	v	v
Amdahl 1200/1400	v	n	n	v	n	n	n	n
Ardent Titan-1	n [c]	n	n	v	v	v	v	v
CDC CYBER 205	v	n	n	v	v	v	v	v
CDC CYBER 990	n	n	n	n	n	n	n	n
Convex C series	v	n	n	v	n	v	v	v
CRAY X-MP CFT	v	n	n	v	n	n	v	v [a]
CRAY X-MP CFT77	v	n	n	v	n [b]	v	v	v
CRAY-2	v	n	n	v	n	n	n	n
ETA-10	v	n	n	v	v	v	v	v
Gould NP1	n	n	n	n	n	n	v	v
Hitachi S-810/820	v	n	n	v	v	v	v	v
IBM 3090/VF	v	n	n	v	n	n	n	n
Intel iPSC-VX	v	n	n	v	n	v	v	v
NEC SX/2	v	n	n	v	v	n	v	n
SCS-40	v	n	n	v	n	n	n	n
Stellar GS1000	v	n	n	v	n	n	v	v
Unisys ISP	v	p	p	v	v	v	v	v

	s151	s152	s161	s171	s172	s173	s174	s175
Alliant FX/8	n [c]	n	n	v	v	v	v	n [c]
Amdahl 1200/1400	v	v	v	v	v	v	n	v
Ardent Titan-1	v	v	n	n	v	v	v	n
CDC CYBER 205	n	n	n	v	v	v	v	n
CDC CYBER 990	n	n	n	v	v	v	v	n
Convex C series	v	v	n	v	v	n	v	n
CRAY X-MP CFT	v [a]	n	n	v	v	v [a]	v [a]	v [a]
CRAY X-MP CFT77	v [b]	n	n	v	v	v [b]	v	v [b]
CRAY-2	n	n	n	v	v	n	n	n
ETA-10	v	v	n	v	v	v	v	n
Gould NP1	n	n	n	n	n	v	v	n
Hitachi S-810/820	n	n	n	n	n	n	v	n
IBM 3090/VF	n	n	n	n	v	n	v	n
Intel iPSC-VX	n	n	n	v	v	v	v	n
NEC SX/2	n	n	n	v	v	n	v	n
SCS-40	n	n	n	v	v	n	n	n
Stellar GS1000	n	n	n	v	v	n	v	n
Unisys ISP	v [b]	n	v	v	v	v	v	v

	s211	s212	s221	s222	s231	s232	s233	s234
Alliant FX/8	p	v	v	p	v	n	p	n [c]
Amdahl 1200/1400	v	v	p	p [d]	v	n	p	n
Ardent Titan-1	v	v	p	p	v	n	v	n
CDC CYBER 205	v	v	v	p	v	n	p	n
CDC CYBER 990	v	v	p	p	v	n	v	n
Convex C series	v	v	p	p	v	n	v	n
CRAY X-MP CFT	n	v	n	n	v	n	p	n
CRAY X-MP CFT77	n	n	n	n	n	n [b]	p [b]	n
CRAY-2	n	n	n	n	v	n	p	n
ETA-10	v	v	p	p	v	n	p	n
Gould NP1	v	v	p	p	v	n	p	n
Hitachi S-810/820	v	v	v	p	v	n	p	n
IBM 3090/VF	v	v	p	p	v	n	v	n
Intel iPSC-VX	v	v	p	p	v	n	p	n
NEC SX/2	v	v	v	p	v	n	p	n
SCS-40	n	v	n	n	v	n	p	n
Stellar GS1000	v	v	p	p	v	n	v	n
Unisys ISP	v	v	p	p	v	n	p	n

	s241	s242	s243	s244	s245	s251	s252	s253
Alliant FX/8	v	n	v	p	v	v	v	v
Amdahl 1200/1400	v	p	v	v	p	v	p [d]	v
Ardent Titan-1	n	p	n	n	n	v	v	n
CDC CYBER 205	v	n	v	p	v	v	v	v
CDC CYBER 990	n	p	n	n	n	n	n	n
Convex C series	v	p	v	v	p	v	v	v
CRAY X-MP CFT	v	n	v	n	n	v	n	n
CRAY X-MP CFT77	n	n	n	n	n	v	v	n
CRAY-2	n	n	n	n	n	v	n	n
ETA-10	v	n	v	p	p	v	v	v
Gould NP1	v	n	v	v	p	v	v	v
Hitachi S-810/820	v	n	v	p	v	v	v	v
IBM 3090/VF	v	p	n	v	n	v	n	v
Intel iPSC-VX	v	n	v	p	p	v	p	v
NEC SX/2	v	p	v	n	v	v	p	v
SCS-40	v	n	v	n	n	v	n	n
Stellar GS1000	v	p	v	v	p	v	p	v
Unisys ISP	n	p	n	n	p	v	p	v

	s254	s255	s271	s272	s273	s274	s275	s276
Alliant FX/8	p	n	v	v	v	v	n	v
Amdahl 1200/1400	p [d]	p	v	v	v	v	n	v
Ardent Titan-1	v	v	v	n	v	v	n	v
CDC CYBER 205	p	n	v	v	v	v	n	v
CDC CYBER 990	v	v	n	n	p	p	n	p
Convex C series	v	v	v	v	v	v	n	v
CRAY X-MP CFT	n	n	v	n	v	v	n	v
CRAY X-MP CFT77	n	n	v	v	v	v	n [b]	n
CRAY-2	n	n	n	n	n	n	n	n
ETA-10	p	n	v	v	v	v	n	v
Gould NP1	p	n	v	v	v	v	n	v
Hitachi S-810/820	v	v	v	v	v	v	n	v
IBM 3090/VF	n	n	v	v	v	v	n	v
Intel iPSC-VX	p	n	v	v	v	v	n	v
NEC SX/2	n	n	v	v	v	v	n	v
SCS-40	n	n	n	n	n	n	n	n
Stellar GS1000	p	p	v	v	v	v	n	v
Unisys ISP	n	n	v	v	v	v	v	v

	s277	s278	s279	s2710	s2711	s2712	s281	s291
Alliant FX/8	n	v	v	v	v	v	n	p
Amdahl 1200/1400	v	v	v	v	v	v	n	p [d]
Ardent Titan-1	n	p	p	v	v	v	n	v
CDC CYBER 205	n	v	v	v	v	v	n	p
CDC CYBER 990	n	p	p	n	n	n	n	n
Convex C series	v	v	v	v	v	v	n	v
CRAY X-MP CFT	n	v	n	n	v	v	n [a]	n
CRAY X-MP CFT77	n	v	v	n	v	n	n	n
CRAY-2	n	n	n	n	n	v	n	n
ETA-10	n	v	v	v	v	v	n	p
Gould NP1	n	v	v	v	v	v	n	n
Hitachi S-810/820	n	v	v	v	v	v	p	v
IBM 3090/VF	v	v	v	v	v	v	n	n
Intel iPSC-VX	n	v	v	v	v	v	n	p
NEC SX/2	n	v	v	v	v	v	p	v
SCS-40	n	n	n	n	n	v	n	n
Stellar GS1000	v	v	v	v	v	v	n	p
Unisys ISP	v	v	v	v	v	v	p	n

	s292	s293	s311	s312	s313	s314	s315	s316
Alliant FX/8	n	n	v	v	v	v	v	v
Amdahl 1200/1400	p [d]	n	v	p [d]	v	v	v	v
Ardent Titan-1	v	n	v	v	v	n	v	n
CDC CYBER 205	n	n	v	v	v	v	v	v
CDC CYBER 990	n	n	v	p	v	n	v	n
Convex C series	v	n	v	v	v	n	v	n
CRAY X-MP CFT	n	n	v	v	v	v	n	v
CRAY X-MP CFT77	n	n	v	v	v	n	n	n
CRAY-2	n	n	v	v	v	v	n	v
ETA-10	n	n	v	v	v	v	v	v
Gould NP1	n	n	v	v	v	v	v	v
Hitachi S-810/820	v	n	v	v	v	v	v	v
IBM 3090/VF	n	n	v	p	v	n	n	n
Intel iPSC-VX	n	n	v	v	v	n	v	n
NEC SX/2	n	n	v	v	v	v	v	v
SCS-40	n	n	v	v	v	n	n	n
Stellar GS1000	p	n	v	p	v	n	n	n
Unisys ISP	n	n	v	v	v	v	v	v

	s317	s318	s321	s322	s323	s331	s332	s341
Alliant FX/8	n	v	v	v	n	v	n	n
Amdahl 1200/1400	n	v	n	n	n	v	v	v
Ardent Titan-1	v	v	n	n	n	v	n	n
CDC CYBER 205	n	v	n	n	n	n	n	n
CDC CYBER 990	n	n	n	n	n	n	n	n
Convex C series	v	v	n	n	n	v	n	v
CRAY X-MP CFT	v	v	n	n	n	n	v	v
CRAY X-MP CFT77	v	v	n	n	n	v	v	n
CRAY-2	v	v	v	n	n	n	n	n
ETA-10	n	v	n	n	n	n	n	n
Gould NP1	n	v	n	n	n	v	n	n
Hitachi S-810/820	v	v	v	n	v	v	v	v
IBM 3090/VF	n	v	n	n	n	n	n	n
Intel iPSC-VX	n	v	n	n	n	v	n	n
NEC SX/2	v	v	v	n	n	n	n	n
SCS-40	v	v	n	n	n	n	n	n
Stellar GS1000	n	v	n	n	n	v	n	n
Unisys ISP	v	v	v	n	n	v	n	p

	s342	s411	s412	s413	s414	s421	s422	s423
Alliant FX/8	n	v	v	v	n [c]	v	v	v
Amdahl 1200/1400	v	n	n	n	n	v	v	n
Ardent Titan-1	n	n	n	n	n	v	v	v
CDC CYBER 205	n	v	v	v	n	v	v	v
CDC CYBER 990	n	n	n	n	n	v	v	n
Convex C series	v	n	n	n	n	v	v	v
CRAY X-MP CFT	v	n	n	n	n	v	v	n
CRAY X-MP CFT77	n	n	n	n	n [b]	v	v [b]	v
CRAY-2	n	n	n	n	n	v	v	n
ETA-10	n	v	v	v	n	v	v	v
Gould NP1	n	v	n	v	n	v	v	v
Hitachi S-810/820	v	n	n	n	n	v	v	n
IBM 3090/VF	n	n	n	n	n	n	n	n
Intel iPSC-VX	n	v	v	v	n	v	v	v
NEC SX/2	n	n	n	n	n	n	n	n
SCS-40	n	n	n	n	n	v	v	n
Stellar GS1000	n	n	n	n	n	n	n	n
Unisys ISP	p	n	n	n	n	n	n	n

	s424	s425	s426	s427	s441	s442	s451	s452
Alliant FX/8	v	v	v	n [c]	v	n	v	v
Amdahl 1200/1400	n	n	n	n	v	n	v	v
Ardent Titan-1	v	v	v	n	n [c]	n [c]	n	v
CDC CYBER 205	v	v	v	v	v	n	v	v
CDC CYBER 990	n	n	n	n	n	n	v	v
Convex C series	v	v	n	n	v	n	v	v
CRAY X-MP CFT	n	n	n	n	n	n	v	v
CRAY X-MP CFT77	v	v	v [b]	v [b]	n	n	v	v
CRAY-2	n	n	n	n	n	n	v	v
ETA-10	v	v	v	n	v	n	v	v
Gould NP1	v	v	v	n	n	n	v	v
Hitachi S-810/820	n	n	n	n	v	n	v	v
IBM 3090/VF	n	n	n	n	v	n	v	v
Intel iPSC-VX	v	v	v	n	v	n	v	v
NEC SX/2	n	n	n	n	v	n	v	v
SCS-40	n	n	n	n	n	n	v	v
Stellar GS1000	n	v	n	n	n	n	v	v
Unisys ISP	n	n	n	n	v	n	v	v

	s461	s471	s481	s482	s4101	s4111	s4112	s4113
Alliant FX/8	n	n	n [c]	n [c]	v	v	v	v
Amdahl 1200/1400	p [d]	v	v	n	v	v	v	v
Ardent Titan-1	v	n	p	n	v	v	v	v
CDC CYBER 205	n	n	n	n	v	v	v	v
CDC CYBER 990	v	v	p	n	v	n	n	n
Convex C series	p	v	n	n	v	v	v	v
CRAY X-MP CFT	n	n	n	v	v	v	v	v
CRAY X-MP CFT77	n	n	v	v	v	v	v	v
CRAY-2	n	n	n	n	v	v	v	v
ETA-10	n	n	n	n	v	v	v	v
Gould NP1	n	n	n	n	v	n	n	n
Hitachi S-810/820	v	n	v	n	v	v	v	v
IBM 3090/VF	n	n	n	n	v	v	v	v
Intel iPSC-VX	n	n	n	n	n	v	v	v
NEC SX/2	v	n	n	n	v	v	v	v
SCS-40	n	n	n	n	v	n	n	n
Stellar GS1000	n	n	n	n	v	v	v	v
Unisys ISP	n	n	n	n	v	v	v	v

	s4114	s4115	s4116	s4117
Alliant FX/8	v	v	n	v
Amdahl 1200/1400	v	v	n	v
Ardent Titan-1	v	p	n	n [c]
CDC CYBER 205	v	n	n	v
CDC CYBER 990	p	n	n	v
Convex C series	v	v	n	v
CRAY X-MP CFT	v	v	n	v
CRAY X-MP CFT77	v	v	n	v
CRAY-2	v	v	n	v
ETA-10	v	v	n	v
Gould NP1	n	n	n	n
Hitachi S-810/820	v	v	v	v
IBM 3090/VF	n	n	n	v [e]
Intel iPSC-VX	v	v	n	v
NEC SX/2	v	v	n	v
SCS-40	v	n	n	n
Stellar GS1000	v	v	p	v
Unisys ISP	v	v	v	v

[a] These loops were conditionally vectorized. A runtime IF-THEN-ELSE test was compiled which executed either a scalar loop or a vectorized loop.

[b] These loops were conditionally vectorized. For loops with ambiguous subscripts a runtime test was compiled which selected a safe vector length.

[c] These loops were parallelized but not vectorized. The compiler generated code to execute these loops in parallel, but no code was generated to vectorize these loops.

[d] These loops were executed in scalar mode. The compiler indicated partial vectorization was possible, but that the overhead was too large.

[e] These loops were executed in scalar mode. The compiler indicated vectorization was possible, but that scalar execution was faster than vector execution.

Appendix B

SOURCE CODE FOR LOOPS USED

```
c**********************************************************
c                                                         *
c         TEST SUITE FOR VECTORIZING COMPILERS            *
c                                                         *
c    Version:   2.0                                       *
c    Date:      3/14/88                                   *
c    Authors:   Original loops from a variety of          *
c               sources. Collection/synthesis by:         *
c                                                         *
c               David Callahan  -  Rice University        *
c               Jack Dongarra   -  Argonne National Lab   *
c               David Levine    -  Argonne National Lab   *
c                                                         *
c    Compile:   Use the compiler options that show the    *
c               automatic vectorization capabilities      *
c               of the compiler.  Do not use compiler     *
c               directives or interactive compilation     *
c               features for additional vectorizations.   *
c               Do not modify the source code. The        *
c               subroutines in section 1.5 may be         *
c               compiled independently.                   *
c                                                         *
c**********************************************************
c**********************************************************
c                                                         *
c                   DEPENDENCE ANALYSIS                   *
c                                                         *
c**********************************************************
c%1.1
      subroutine s111(a,n)
c
c     linear dependence testing
c     dependence testing - vectorizable
c
      dimension a(1000)
      do 400 i = 2,100,2
         a(i) = a(i-1)
  400 continue
      write (unit=6,fmt=100) a(n)
  100 format (e12.6)
      return
      end
c%1.1
      subroutine s112(a,n)
c
c     linear dependence testing
c     loop reversal
c
      dimension a(1000)
      do 700 i = 999,1,-1
         a(i+1) = a(i)
  700 continue
      write (unit=6,fmt=100) a(n)
  100 format (e12.6)
      return
      end
c%1.1
      subroutine s113(a,n)
c
c     linear dependence testing
c     a(i)=a(1) but no actual dependence cycle
c
      integer n
      real a(*)
      do 610 i = 2,n
         a(i) = a(1)
  610 continue
      return
      end
c%1.1
      subroutine s114(aa,bb,n)
c
c     linear dependence testing
c     transpose vectorization
c
      integer n
      real aa(n,*),bb(n,*)
      do 300 j = 1,n
         do 300 i = 1,j-1
            aa(i,j) = aa(j,i) + bb(i,j)
  300 continue
      return
      end
c%1.1
      subroutine s115(aa,n)
c
c     linear dependence testing
c     lower triangular system
c
c
      integer n
      real aa(n,*)
      do 320 j = 1,n
         do 320 k = 1,j-1
            do 320 i = k+1,n
  320 aa(i,j) = aa(i,j) + aa(i,k) * aa(k,j)
      return
      end
c%1.1
      subroutine s116(a,b,n)
c
c     linear dependence testing
c
      integer n
      real a(*),b(*)
      do 450 i = 1,n,5
         a(i) = a(i+1) + a(i)*b(i)
         a(i+1) = a(i+2) + a(i+1)*b(i+1)
```

```
         a(i+2) = a(i+3) + a(i+2)*b(i+2)
         a(i+3) = a(i+4) + a(i+3)*b(i+3)
         a(i+4) = a(i+5) + a(i+4)*b(i+4)
  450 continue
      return
      end
c%1.1
      subroutine s117(a,logn,n)
c
c     linear dependence testing
c     fft subscripting
c
      integer n,logn
      real a(*)
      j = 1
      do 970 i = 1,logn
         do 971 k = 1,n,j
            do 971 l = 1,j
               t = a(k+l-1) + a(k+l+j-1)
               u = a(k+l-1) - a(k+l+j-1)
               a(k+l-1) = t
               a(k+l+j-1) = u
  971    continue
         j = j * 2
  970 continue
      return
      end
c%1.2
      subroutine s121
c
c     induction variable recognition
c     loop with possible ambiguity because of scalar store
c
      parameter ( n = 1000 )
      real a(1000)
      integer i, j
      do 70, i = 1, n-1
         j = i+1
         a(i) = a(j)
70    continue
      write(unit=6,fmt=11) (a(i),i=1,n)
   11 format(e12.6)
      return
      end
c%1.2
      subroutine s122(xx,yy,n)
c
c     induction variable recognition
c     mixed variable and constant lb, ub, and stride
c@    k is not initialized before first use.
c@    n1,n3 are not defined.
c
      dimension xx(100), yy(100), zz(100)
      xx(1) = 6.
      j = 6
      do 60 i=n1,100,n3
         xx(i)=yy(i)*zz(100-k+1)
         k = k + j
   60 continue
      write (unit=6,fmt=11) j, xx(n)
   11 format (i6,e12.6)
      return
      end
c%1.2
      subroutine s123(a,b,c,n)
c
c     induction variable under an if
c
      integer n
      real a(*),b(*),c(*)
      j = 0
      do 50 i = 1,n
         j = j + 1
         a(j) = b(i)
         if(c(i).gt.0) then
            j = j + 1
            a(j) = c(i)
         endif
   50 continue
      return
      end
c%1.2
      subroutine s124(a,b,c,n)
c
c     induction variable recognition
c     induction variable under both sides of if (same value)
c
      integer n
      real a(*),b(*),c(*)
      j = 0
      do 60 i = 1,n
         if(b(i).gt.0) then
            j = j + 1
            a(j) = b(i)
            else
            j = j + 1
            a(j) = c(i)
         endif
   60 continue
      return
      end
c%1.2
      subroutine s125(a,bb,cc,n)
c
c     induction variable recognition
c     induction variable in two loops; collapsing possible
c
      integer n
      real a(*),bb(n,*),cc(n,*)
      k = 0
      do 20 i = 1,n
         do 20 j = 1,n
            k = k + 1
            a(k) = bb(i,j) + cc(i,j)
   20 continue
      return
      end
c%1.2
```

```
      subroutine s126(a,bb,n)
c
c     induction variable recognition
c     induction variable in two loops
c     recurrence in inner loop
c
      integer n
      real a(*),bb(n,*)
      k = 1
      do 40 i = 1,n
         do 41 j = 2,n
            bb(i,j) = bb(i,j-1) + a(k)
            k = k + 1
   41    continue
         k = k + 1
   40 continue
      return
      end
c%1.2
      subroutine s127(a,b,c,n)
c
c     induction variable recognition
c     induction variable with multiple increments
c
      integer n
      real a(*),b(*),c(*)
      j = 0
      do 10 i = 1,n
         j = j + 1
         a(j) = b(i)
         j = j + 1
         a(j) = c(i)
   10 continue
      return
      end
c%1.3
      subroutine s131(a,b,n)
c
c     global data flow analysis
c     forward substitution
c
      integer n
      real a(*),b(*)
      m = 1
      if(a(1).gt.0)then
         a(1) = b(1)
      endif
      do 340 i = 2,n-1
         a(i) = a(i+m) + b(i)
  340 continue
      return
      end
c%1.3
      subroutine s132
c
c     global data flow analysis
c     loop with multiple dimension ambiguous subscripts
c
      parameter ( n = 100 )
      real a(100,100)
      integer i, j, k, m
      m = 1
      j = m
      k = m+1
      do 170, i=2,n
         a(i,j) = a(i-1,k)*3.5
170   continue
      write(unit=6,fmt=11) ((a(i,j),i=1,n),j=1,n)
   11 format(e12.6)
      return
      end
c%1.5
      subroutine s151(a,b,n)
c
c     interprocedural data flow analysis
c     universal compilation - passing parameter
c     information into a subroutine
c
      integer n
      real a(*),b(*)
      call s151s(a,b,n-1,1)
      return
      end
      subroutine s151s(a,b,n,m)
      integer n,m
      real a(*),b(*)
      do 730 i = 1,n
         a(i) = a(i+m) + b(i)
  730 continue
      return
      end
c%1.5
      subroutine s152(a,b,n)
c
c     interprocedural data flow analysis
c     universal compilation
c     collecting info from subroutine
c
      integer n
      real a(*),b(*)
      do 750 i = 1,n
         b(i) = b(i) + 2
         call s152s(a,b,i)
  750 continue
      return
      end
      subroutine s152s(a,b,i)
      integer i
      real a(*),b(*)
      a(i) = a(i) + b(i)
      return
      end
c%1.6
      subroutine s161(a,b,x,n)
c
c     control flow
c     tests for recognition of loop independent dependences
c     between statements in mutually exclusive regions.
```

```
c@    array bounds error when i = 100.
c
      dimension a(100),b(100),x(100)
      do 30 i = 1,100
        if (b(i).lt.0) go to 10
        a(i) = x(i)
        go to 30
   10   x(i+1) = a(i)
   30 continue
      write(unit=6,fmt=11) a(n),b(n),x(n)
   11 format(3e12.6)
      return
      end
c%1.7
      subroutine s171(a,b,n)
c
c     symbolics
c     symbolic dependence tests
c
      integer n
      real a(*),b(*)
      do 1030 i = 1,n
         a(i*n) = a(i*n) + b(i)
 1030 continue
      return
      end
c%1.7
      subroutine s172(a,b,n)
c
c     symbolics
c     symbolics - not vectorizable
c@    could be vectorized if you assume n3 .NE. 0,
c@    a reasonable assumption.
c@    n1,n2,n3 are not defined.
c
      dimension a(1000),b(1000)
      do 500 i = n1,n2,n3
         a(i) = a(i) + b(i)
  500 continue
      write (unit=6,fmt=100) a(n)
  100 format (e12.6)
      return
      end
c%1.7
      subroutine s173(a,b,n)
c
c     symbolics
c     expression in loop bounds and subscripts
c
      integer n
      real a(*),b(*)
      do 370 i = 1,n/2
         a(i+n/2) = a(i) + b(i)
  370 continue
      return
      end
c%1.7
      subroutine s174
c
c     symbolics
c     loop with subscript that may seem ambiguous
c
      parameter ( n = 1000 )
      real a(1000)
      integer i
      do 50, i= 1, n/2
         a(i) = a( i + n/2 )
50    continue
      write(unit=6,fmt=11) (a(i),i=1,n)
   11 format(e12.6)
      return
      end
c%1.7
      subroutine s175(a,b,n,inc)
c
c     symbolics
c     symbolic dependence tests
c
      integer n,inc
      real a(*),b(*)
      do 1020 i = 1,n,inc
         a(i) = a(i+inc) + b(i)
 1020 continue
      return
      end
c
c**********************************************************
c                                                         *
c                     VECTORIZATION                       *
c                                                         *
c**********************************************************
c%2.1
      subroutine s211(a,b,c,n)
c
c     statement reordering
c     statement reordering allows vectorization
c
      integer n
      real a(*),b(*),c(*)
      do 270 i = 2,n-1
         a(i) = b(i-1) + c(i)
         b(i) = b(i+1) - 2.
  270 continue
      return
      end
c%2.1
      subroutine s212
c
c     statement reordering
c     dependency needing temporary
c
      parameter ( n = 1000 )
      real a(1000), b(1000), x(6000)
      integer i
      do 20, i=1,n-1
         a(i) = x(i)
         b(i) = b(i)+a(i+1)
20    continue
```

```
      write(unit=6,fmt=11) (a(i),i=1,n)
      write(unit=6,fmt=11) (b(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.2
      subroutine s221
c
c     loop distribution
c     loop that is partially recursive
c
      parameter ( n = 1000 )
      real a(1000), b(1000), x(6000), y(6000)
      integer i
      do 80, i = 2 , n
         a(i) = a(i) + ( x(i) * y(i) )
         b(i) = b(i-1) + a(i) + y(i)
80    continue
      write(unit=6,fmt=11) (a(i),i=1,n)
      write(unit=6,fmt=11) (b(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.2
      subroutine s222(a,b,c,n)
c
c     loop distribution
c     partial loop vectorization, recurrence in middle
c
      integer n
      real a(*),b(*),c(*)
      do 240 i = 2,n
         a(i) = a(i) + b(i)
         b(i) = b(i-1)*b(i-1)*a(i)
         a(i) = a(i) - b(i)
  240 continue
      return
      end
c%2.3
      subroutine s231
c
c     loop interchange
c     loop with multiple dimension recursion
c
      parameter ( n = 100 )
      real a(100,100), b(100,100)
      integer i, j
      do 160, i=1,n
         do 160, j=2,n
            a(i,j) = a(i,j-1)+b(i,j)
160   continue
      write(unit=6,fmt=11) ((a(i,j),i=1,n),j=1,n)
   11 format(e12.6)
      return
      end
c%2.3
      subroutine s232(aa,bb,n)
c
c     loop interchange
c     interchanging of triangular loops
c
      integer n
      real aa(n,*),bb(n,*)
      do 290 j = 2,n
         do 290 i = 2,j
            aa(i,j) = aa(i-1,j)*aa(i-1,j)+bb(i,j)
  290 continue
      return
      end
c%2.3
      subroutine s233(aa,bb,cc,n)
c
c     loop interchange
c     interchanging with one of two inner loops
c
      integer n
      real aa(n,*),bb(n,*),cc(n,*)
      do 840 i = 2,n
         do 841 j = 2,n
            aa(i,j) = aa(i,j-1) + cc(i,j)/aa(i,j-1)
  841    continue
         do 842 j = 2,n
            bb(i,j) = bb(i-1,j) + cc(i,j)/bb(i-1,j)
  842    continue
  840 continue
      return
      end
c%2.3
      subroutine s234(aa,bb,cc,n)
c
c     loop interchange
c     if loop to do loop
c     interchanging with if loop necessary
c
      integer n
      real aa(n,*),bb(n,*),cc(n,*)
      i = 1
  232 if(i.gt.n) goto 231
      j = 3
  233 if(j.gt.n) goto 230
      aa(i,j) = aa(i,j-1)*bb(i,j-1)
      bb(i,j) = aa(i,j-1)+bb(i,j-2)
      cc(i,j) = cc(i,j-1)*cc(i,j-2)
      j = j + 1
      goto 233
  230 i = i + 1
      goto 232
  231 continue
      return
      end
c%2.4
      subroutine s241(a,b,c,n)
c
c     node splitting
c     preloading necessary to allow vectorization
c
      integer n
      real a(*),b(*),c(*)
```

```
      do 280 i = 2,n-1
         a(i) = b(i) + c(i)
         b(i) = a(i) + a(i+1)
  280 continue
      return
      end
c%2.4
      subroutine s242(a,b,c,n)
c
c     node splitting
c@    array bounds error when i=1, i=100
c
      dimension a(100),b(100),c(100),d(100)
      dimension abc1(100),abc2(100),bcd1(100)
      do 510 i = 1,100
         abc1(i-1) = abc2(i-1) + 1
         bcd1(i-1) = abc1(i-1) ** 2
         a(i) = bcd1(i-1) + d(i)
         abc2(i) = b(i+1) + b(i-1)
         b(i) = c(i) + 1
         c(i+1) = b(i) + 1
  510 continue
      write(unit=6,fmt=12) c(n)
      write(unit=6,fmt=12) b(n)
   12 format (e12.6)
      return
      end
c%2.4
      subroutine s243(a,b,c,n)
c
c     node splitting
c     false dependence cycle breaking
c
      integer n
      real a(*),b(*),c(*)
      do 630 i = 2,n-1
         a(i) = b(i) + c(i)
         b(i) = a(i) - 2.
         a(i) = b(i) + a(i+1)
  630 continue
      return
      end
c%2.4
      subroutine s244(a,b,c,n)
c
c     node splitting
c     false dependence cycle breaking
c
      integer n
      real a(*),b(*),c(*)
      do 640 i = 2,n-1
         a(i) = b(i) + c(i)
         b(i) = c(i) - 2.
         a(i+1) = b(i) + a(i+1)
  640 continue
      return
      end
c%2.4
      subroutine s245(a,b,c,n)
c
c     node splitting
c     array bounds error when i=1
c
      dimension a(100),b(100),c(100),d(100)
      t2 = 0
      do 800 i = 1,100
         a(i) = a(i-1) + t1 + t2 + b(i) + c(i) + d(i)
  800 continue
      write(unit=6,fmt=12) a(n)
   12 format (e12.6)
      return
      end
c%2.5
      subroutine s251(b,c,n)
c
c     scalar and array expansion
c     scalar expansion
c
      dimension b(1000),c(1000),d(1000)
      do 900 i = 1,100,1
         abc = b(i) * c(i)
         d(i) = abc * abc
  900 continue
      write (unit=6,fmt=100) d(n)
  100 format (e12.6)
      return
      end
c%2.5
      subroutine s252
c
c     scalar and array expansion
c     loop with ambiguous scalar temporary
c
      parameter ( n = 6000 )
      real a(6000), x(6000), y(6000), s, t
      integer i
      t = 0.
      do 40, i=1,n
         s = x(i) * y(i)
         a(i) = s + t
         t = s
40    continue
      write(unit=6,fmt=11) (a(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.5
      subroutine s253(a,b,c,n)
c
c     scalar and array expansion
c     scalar expansion, assigned under if
c
      integer n
      real a(*),b(*),c(*)
      do 760 i = 1,n
         if(a(i).gt.b(i))then
            t = a(i) - b(i)
            c(i) = c(i) + t
```

```
            a(i) = t
          endif
  760 continue
      return
      end
c%2.5
      subroutine s254(a,b,n)
c
c     scalar and array expansion
c     carry around variable
c
      integer n
      real a(*),b(*)
      x = a(n)
      do 390 i = 1,n
         b(i) = (a(i) + x) / 2.
         x = a(i)
  390 continue
      return
      end
c%2.5
      subroutine s255(a,b,n)
c
c     scalar and array expansion
c     carry around variables, 2 levels
c
      integer n
      real a(*),b(*)
      x = a(n)
      y = a(n-1)
      do 400 i = 1,n
         b(i) = (a(i) + x + y) / 3.
         y = x
         x = a(i)
  400 continue
      return
      end
c%2.7
      subroutine s271
c
c     control flow
c     loop with singularity handling
c
      parameter ( n = 1000 )
      real a(1000), y(6000), z(6000)
      integer i
      do 140, i=1,n
         if (y(i).gt.0.) a(i) = y(i)/z(i)
140   continue
      write(unit=6,fmt=11) (a(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.7
      subroutine s272
c
c     control flow
c     loop with independent conditional
c
      parameter ( n = 1000 )
      real a(1000), x(6000), y(6000), z(6000)
      real s, r, t
      integer i
      t = 1.
      do 100, i = 1, n
         if (z(i) .ge. t) then
            s = x(i) * y(i) + 3.1
            r = x(i) + y(i) * 2.9
            a(i) = sqrt(s**2*r)
         endif
100   continue
      write(unit=6,fmt=11) (a(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.7
      subroutine s273
c
c     control flow
c     simple loop with dependent conditional
c
      parameter ( n = 4000 )
      real a(4000), b(4000), c(4000)
      real x(6000), y(6000), z(6000)
      integer i
      do 120, i = 1 , n
         a(i) = a(i) + y(i) + z(i)
         if (a(i) .lt. 0.) b(i) = b(i) + x(i) + y(i)
         c(i) = c(i) + a(i) + x(i)
120   continue
      write(unit=6,fmt=11) (a(i),i=1,n)
      write(unit=6,fmt=11) (b(i),i=1,n)
      write(unit=6,fmt=11) (c(i),i=1,n)
   11 format(e12.6)
      return
      end
c%2.7
      subroutine s274
c
c     control flow
c     complex loop with dependent conditional
c
      parameter ( n = 1000 )
      real a(1000), b(1000)
      real x(6000), y(6000), z(6000)
      integer i
      do 130, i = 1 , n
         a(i) = x(i) + z(i)
         if (a(i) .eq. 0.) then
            b(i) = a(i) * b(i)
         else
            a(i) = y(i) * z(i)
            b(i) = 1.
         endif
130   continue
      write(unit=6,fmt=11) (a(i),i=1,n)
      write(unit=6,fmt=11) (b(i),i=1,n)
   11 format(e12.6)
```

```
      return
      end
c%2.7
      subroutine s275(aa,bb,n)
c
c     control flow
c     if around inner loop, interchanged needed
c
      integer n
      real aa(n,*),bb(n,*)
      do 480 i = 2,n
         if(bb(i,1).gt.0)then
            do 481 j = 2,n
  481       bb(i,j) = bb(i,j)/bb(i,j-1)
         endif
  480 continue
      return
      end
c%2.7
      subroutine s276(a,b,c,n)
c
c     control flow
c     ifs which trim the index set
c
      integer n
      real a(*),b(*),c(*)
      do 660 i = 1,n
         a(i) = b(i) + c(i)
         if(i.gt.5) b(i) = abs(b(i))
         if(i.lt.99) a(i) = -a(i)
  660 continue
      return
      end
c%2.7
      subroutine s277(a,b,x,n)
c
c     control flow
c     test for dependences arising
c     from guard variable computation
c@    y is used before being defined
c@    array bounds error when i = 100.
c
      dimension a(100),b(100),x(100),y(100)
      do 130 i = 1,100
        if (a(i).ge.0) go to 120
        if (b(i).ge.0) go to 110
        a(i) = x(i)
  110   continue
        b(i+1) = y(i)
  120   continue
  130 continue
      write(unit=6,fmt=11) a(n),b(n),x(n)
   11 format(3e12.6)
      return
      end
c%2.7
      subroutine s278(a,b,c,n)
c
c     control flow
c     if/goto to block if-then-else
c
      integer n
      real a(*),b(*),c(*)
      do 120 i = 1,n
         if(a(i).gt.0)goto 121
         b(i) = -b(i)
         goto 122
  121    continue
         c(i) = -c(i)
  122    continue
         a(i) = b(i) + c(i)
  120 continue
      return
      end
c%2.7
      subroutine s279(a,b,c,n)
c
c     control flow
c     vector if/gotos
c
      integer n
      real a(*),b(*),c(*)
      do 810 i = 1,n
         if(a(i).gt.0)goto 811
         b(i) = -b(i)
         if(abs(b(i)).le.a(i))goto 812
         c(i) = abs(c(i))
         goto 812
  811    continue
         c(i) = -c(i)
  812    continue
         a(i) = b(i) + c(i)
  810 continue
      return
      end
c%2.7
      subroutine s2710(a,b,c,x,n)
c
c     control flow
c     scalar and vector ifs
c
      integer n
      real a(*),b(*),c(*),x
      do 790 i = 1,n
         if(a(i).gt.b(i))then
            a(i) = a(i) - b(i)
            if(n.gt.10)then
               c(i) = abs(c(i))
               else
               c(i) = 0.
            endif
            else
            b(i) = a(i)
            if(x.gt.0)then
               c(i) = a(i)
               else
               c(i) = -c(i)
            endif
```

```
        endif
  790 continue
      return
      end
c%2.7
      subroutine s2711(a,b,c,n)
c
c     control flow
c     semantic if removal
c
      integer n
      real a(*),b(*),c(*)
      do 650 i = 1,n
         if(a(i).ne.0) b(i) = b(i) + a(i) * c(i)
  650 continue
      return
      end
c%4.5
      subroutine s2712(a,b,n)
c
c     control flow
c     if to elemental min
c
      integer n
      real a(*),b(*)
      do 70 i = 1,n
         if(a(i).gt.b(i)) a(i) = b(i)
   70 continue
      return
      end
c%2.8
      subroutine s281(a,b,n)
c
c     crossing thresholds ( index set splitting )
c     index set splitting
c
      integer n
      real a(*),b(*)
      do 990 i = 1,n
         x = a(n-i+1) + b(i)
         a(i) = x + 3.
         b(i) = x
  990 continue
      return
      end
c%2.9
      subroutine s291(a,b,n)
c
c     loop peeling
c     wrap around variable, 1 level
c
      integer n
      real a(*),b(*)
      im1 = n
      do 410 i = 1,n
         b(i) = (a(i) + a(im1)) / 2.
         im1 = i
  410 continue
      return
      end
c%2.9
      subroutine s292(a,b,n)
c
c     loop peeling
c     wrap around variable, 2 levels
c
      integer n
      real a(*),b(*)
      im1 = n
      im2 = n-1
      do 420 i = 1,n
         b(i) = (a(i) + a(im1) + a(im2)) / 3.
         im2 = im1
         im1 = i
  420 continue
      return
      end
c%2.9
      subroutine s293(a,n)
c
c     loop peeling
c     a(i)=a(1) with actual dependence cycle
c@    above comment misleading, loop is vectorizable
c
      integer n
      real a(*)
      do 620 i = 1,n
         a(i) = a(1)
  620 continue
      return
      end
c
c**********************************************************
c                                                        *
c                  IDIOM RECOGNITION                     *
c                                                        *
c**********************************************************
c%3.1
      subroutine s311(a,b,x,n)
c
c     reductions
c     sum reduction
c
      integer n
      real a(*),b(*),x
      do 850 i = 1,n
         x = x + a(i)
         b(i) = a(i) + 2.
  850 continue
      return
      end
c%3.1
      subroutine s312(a,b,x,n)
c
c     reductions
c     product reduction
c
      integer n
```

```
      real a(*),b(*),x
      do 860 i = 1,n
         x = x * a(i)
         b(i) = a(i) + 2.
  860 continue
      return
      end
c%3.1
      subroutine s313(a,b,n)
c
c     reductions
c     dot product
c
      dimension a(1000),b(1000)
      s = 0.
      do 930 i = 1,n
         s = s + a(i) * b(i)
  930 continue
      write (unit=6,fmt=100) s
  100 format (e12.6)
      return
      end
c%3.1
      subroutine s314(x,b,n)
c
c     reductions
c     if to max reduction
c
      integer n
      real x,b(*)
      do 80 i = 1,n
         if(b(i).gt.x) x = b(i)
   80 continue
      return
      end
c%3.1
      subroutine s315(x,j,b,n)
c
c     reductions
c     if to max with index reduction, 1 dimension
c
      integer n,j
      real x,b(*)
      do 90 i = 1,n
         if(b(i).gt.x)then
            x = b(i)
            j = i
         endif
   90 continue
      return
      end
c%3.1
      subroutine s316(a,n)
c
c     reductions
c     minval
c
      dimension a(1000)
      s = a(1)
```

```
      do 960 i = 2,n
         if (a(i) .lt. s) s = a(i)
  960 continue
      write (unit=6,fmt=100) s
      write (unit=6,fmt=100) a(n)
  100 format (e12.6)
      return
      end
c%3.1
      subroutine s317(n)
c
c     reductions
c     tests scalar expansion.  From a benchmark to test
c     the scalar speed of a machine. The best results
c     are fully vectorized. the compiler expands .995
c     into a vector of .995 and does a product reduction
c     on the vector of .995
c@    this loop has closed form solution: q = .995**n
c
      q = 1.
      do 50, i = 1,n
         q = .995*q
   50 continue
      write(unit=6,fmt=11) q
   11 format(e12.6)
      return
      end
c%3.1
      subroutine s318(a,b,x,n)
c
c     reductions
c
      integer n
      real a(*),b(*),x
      do 430 i = 1,n,5
         x = x + a(i)*b(i) + a(i+1)*b(i+1) + a(i+2)*b(i+2)
     $   + a(i+3)*b(i+3) + a(i+4)*b(i+4)
  430 continue
      return
      end
c%3.2
      subroutine s321(a,b,n)
c
c     recurrences
c     first order linear recurrence
c
      integer n
      real a(*),b(*)
      do 870 i = 2,n
         a(i) = a(i) + a(i-1)*b(i)
  870 continue
      return
      end
c%3.2
      subroutine s322(a,b,c,n)
c
c     recurrences
c     second order linear recurrence
c
```

```
      integer n
      real a(*),b(*),c(*)
      do 880 i = 3,n
         a(i) = a(i) + a(i-1)*b(i) + a(i-2)*c(i)
  880 continue
      return
      end
c%3.2
      subroutine s323(a,b,c,d,n)
c
c     recurrences
c     coupled recurrence
c@    array bounds error when i = 1.
c
      integer n
      real a(*),b(*),c(*),d(*)
      do 1040 i = 1,n
         a(i) = b(i-1) + c(i)
         b(i) = a(i) + d(i)
 1040 continue
      return
      end
c%3.3
      subroutine s331(a,j,n)
c
c     search loops
c     if to last-1
c
      integer j,n
      real a(*)
      do 130 i = 1,n
         if(a(i).eq.0) j = i
  130 continue
      return
      end
c%3.3
      subroutine s332(a,b,j,n)
c
c     search loops
c     search loop saving index
c
      integer n,j
      real a(*),b(*)
      do 510 i = 1,n
         if(a(i).gt.b(i))then
            j = i
            goto 511
         endif
  510 continue
      j = 0
      return
  511 continue
      return
      end
c%3.4
      subroutine s341(a,b,j,n)
c
c     packing
c
      integer n,j
      real a(*),b(*)
      j = 0
      do 900 i = 1,n
         if(a(i).gt.0)then
            j = j + 1
            b(j) = a(i)
         endif
  900 continue
      return
      end
c%3.4
      subroutine s342(a,b,j,n)
c
c     packing
c     unpacking
c
      integer n,j
      real a(*),b(*)
      j = 0
      do 910 i = 1,n
         if(a(i).gt.0)then
            j = j + 1
            a(i) = b(j)
         endif
  910 continue
      return
      end
c
c**********************************************************
c                                                         *
c               LANGUAGE COMPLETENESS                     *
c                                                         *
c**********************************************************
c%4.1
      subroutine s411(a,b,c,n)
c
c     loop recognition
c     if loop to do loop, zero trip
c
      integer n
      real a(*),b(*),c(*)
      i = 0
  140 continue
      i = i + 1
      if(i.gt.n)goto 141
      a(i) = b(i) + c(i)
      goto 140
  141 continue
      return
      end
c%4.1
      subroutine s412(a,b,c,n,inc)
c
c     loop recognition
c     if loop with variable increment
c
      integer n,inc
      real a(*),b(*),c(*)
```

```
      i = 0
  950 continue
      i = i + inc
      if(i.gt.n)goto 951
      a(i) = b(i) + c(i)
      goto 950
  951 continue
      return
      end
c%4.1
      subroutine s413(a,b,c,n)
c
c     loop recognition
c     if loop to do loop, code on both sides of increment
c
      integer n
      real a(*),b(*),c(*)
      i = 0
  180 continue
      if(i.gt.n)goto 181
      b(i) = abs(b(i))
      i = i + 1
      a(i) = c(i)
      goto 180
  181 continue
      return
      end
c%4.1
      subroutine s414(aa,bb,cc,n)
c
c     loop recognition
c     if loop to do loop, interchanging with do necessary
c
      integer n
      real aa(n,*),bb(n,*),cc(n,*)
      i = 1
  222 if(i.gt.n) goto 221
      do 220 j = 3,n
         aa(i,j) = aa(i,j-1)*bb(i,j-1)
         bb(i,j) = aa(i,j-1)+bb(i,j-2)
  220 cc(i,j) = cc(i,j-1)*cc(i,j-2)
      i = i + 1
      goto 222
  221 continue
      return
      end
c%4.2
      subroutine s421(n)
c
c     storage classes and equivalencing
c     equivalence- no overlap
c@    array bounds error when i = 100.
c
      dimension xx(100), yy(100)
      equivalence (xx(1),yy(1))
      do 205 i = 1,100
         xx(i) = yy(i+1)
  205 continue
      write(unit=6,fmt=100) xx(n)
  100 format(e12.6)
      return
      end
c%4.2
      subroutine s422(n)
c
c     storage classes and equivalencing
c     equivalence-  vectorizable
c@    array bounds error when i,j
c@    are simultaneously > 96.
c
      dimension x(100,100),y(100,100)
      equivalence (x(1,1),y(1,1))
      do 410 i = 1,100
         do 420 j = 1,100
            x(j+4,i+4) = y(j,i)
  420    continue
  410 continue
      write(unit=6,fmt=100) x(n,n)
  100 format(e12.6)
      return
      end
c%4.2
      subroutine s423(n)
c
c     storage classes and equivalencing
c     common and equivalenced variables - no overlap
c@    misleading comment, there is an anti-dependence
c
      dimension cc(100)
      common /com1/ii1
      common /com2/aa(200)
      equivalence (aa(50),cc(1))
      do 116 ii1 = 1,100
         aa(ii1+1) = cc(ii1)
  116 continue
      write(unit=6,fmt=100) aa(n)
  100 format(e12.6)
      return
      end
c%4.2
      subroutine s424(n)
c
c     storage classes and equivalencing
c     common and equivalenced variables - no overlap
c@    threshold is 100 >= loop upper bound => no dependence
c
      dimension cc(100)
      common /com1/ii1
      common /com2/aa(200)
      equivalence (aa(50),cc(1))
      do 118 ii1 = 1,100
         cc(ii1+50) = aa(ii1-1)
  118 continue
      write(unit=6,fmt=100) cc(n)
  100 format(e12.6)
      return
      end
c%4.2
```

```
      subroutine s425(n)
c
c     storage classes and equivalencing
c     common and equivalence statement
c@    anti-dependence with threshold of 4
c
      dimension b(1000)
c     common /comm1/cc1(100),cc2(100)
      common /comm1/cc1(100),cc2(100),cc3(100),cc4(100,100)
      dimension eqv1(100),eqv2(90)
      equivalence (eqv1(1),cc2(1))
      equivalence (eqv2(1),cc1(5))
      do 920 i = 1,85
         eqv2(i) = cc1(i+8) + b(i)
920   continue
      write(6,100) eqv2(n)
100   format(e12.6)
      return
      end
c%4.2
      subroutine s426(n,y)
c
c     storage classes and equivalencing
c     common and equivalence statement
c@    anti dependence with distance vector <2,6>
c@    vectorizable with respect to both loops
c
      dimension y(100,100)
      common /comm1/cc1(100),cc2(100),cc3(100),cc4(100,100)
      dimension eqv1(100),eqv2(90),eqv3(100),eqv4(100,99)
      equivalence (eqv1(1),cc2(1))
      equivalence (eqv2(1),cc1(5))
      equivalence (eqv3(5),cc3(5))
      equivalence (eqv4(1,2),cc4(1,1))
      do 940 j = 2,80
         do 950 i = 1,90
            eqv4(i,j) = cc4(i+2,j+5) + y(i,j)
  950    continue
  940 continue
      write(6,100) eqv4(n,n)
  100 format(e12.6)
      return
      end
c%4.2
      subroutine s427(n)
c
c     storage classes and equivalencing
c     common and equivalenced variables - overlap
c@    a partially negative test,
c@    vectorizable in chunks of <=50
c
      dimension c(100)
      common /com1/i
      common /com2/a(200)
      equivalence (a(50),c(1))
      do 115 i = 1,100
         c(i+1) = a(i)
  115 continue
      write(unit=6,fmt=100) c(n)
  100 format(e12.6)
      return
      end
c%4.4
      subroutine s441(a,c,n)
c
c     non-logical if's
c     arithmetic if
c@    xx is used before being defined.
c
      dimension a(1000),b(1000),c(1000),d(1000),xx(100)
      do 710 i = 1,100
         if (d(i)) 720,730,740
720      c(i) = a(i)
         goto 750
730      c(i) = b(i)
         goto 750
740      c(i) = xx(i)
750      continue
710   continue
      write(6,100) c(n)
100   format(e12.6)
      return
      end
c%4.4
      subroutine s442(a,c,n)
c
c     non-logical if's
c     computed goto
c@    aa is used before being defined.
c@    ii is used before being defined.
c
      dimension a(1000),b(1000),c(1000)
      dimension d(1000),aa(200),ii(1000)
      do 810 i = 1,100
         goto (815,820,830,840) ii(i)
815      c(i) = aa(i)
         goto 850
820      c(i) = a(i)
         goto 850
830      c(i) = b(i)
         goto 850
840      c(i) = d(i)
850      continue
810   continue
      write(6,100) c(n)
100   format(e12.6)
      return
      end
c%4.5
      subroutine s451(a,b,c,n)
c
c     intrinsic functions
c     intrinsics
c
      integer n
      real a(*),b(*),c(*)
      do 930 i = 1,n
         a(i) = sin(b(i)) + cos(c(i))
```

```
  930 continue
      return
      end
c%4.5
      subroutine s452(a,b,c,m)
c
c     intrinsic functions
c     seq function
c
      dimension a(1000),b(1000),c(1000)
      do 1250 i = 1,m
         a(i) = b(i) + c(i) + i
 1250 continue
      return
      end
c%4.6
      subroutine s461(a,c,n)
c
c     i/o statements
c     write statement
c
      dimension a(1000),b(1000),c(1000),d(1000)
      do 600 i = 1,100
         b(i) = d(i)
         c(i) = a(i) + b(i)
         write(6,100) c(i)
600   continue
      write(6,100) b(n)
100   format(e12.6)
      return
      end
c%4.7
      subroutine s471(z,n)
c
c     call statements
c
      dimension x(100,100),z(100,100),w(100,100),y(100,100)
      equivalence (x(1,1),y(1,1))
      do 510 i = 1,100
         do 520 j = 1,100
            x(j,i) = z(j,i)
            call sub2
            z(j,i) = w(j,i)
  520    continue
  510 continue
      write(unit=6,fmt=100) z(n,n)
  100 format(e12.6)
      return
      end
c%4.8
      subroutine s481(a,b,c,n)
c
c     non-local goto's
c     stop statement
c
      dimension a(1000),b(1000),c(1000)
      do 110 i = 1,100
         if (a(i) .lt. 0.) stop 'stop 1'
         b(i) = c(i)
  110 continue
      write (unit=6,fmt=100) b(n)
  100 format(e12.6)
      return
      end
c%4.8
      subroutine s482(a,b,c,n)
c
c     non-local goto's
c     other loop exit with code before exit
c
      integer n
      real a(*),b(*),c(*)
      do 520 i = 1,n
         c(i) = a(i)
         if(a(i).gt.b(i))goto 521
  520 continue
      return
  521 continue
      return
      end
c%4.10
      subroutine s4101(a,b,c,n)
c
c     data types
c     complex arithmetic
c
      integer n
      complex a(*),b(*),c(*)
      do 920 i = 1,n
         c(i) = a(i) + b(i)
  920 continue
      return
      end
c%4.11
      subroutine s4111(a,b,ip,n)
c
c     indirect addressing
c     indirect addressing on lhs
c
      integer n,ip(*)
      real a(*),b(*)
      do 560 i = 1,n
         a(ip(i)) = b(i)
  560 continue
      return
      end
c%4.11
      subroutine s4112(a,b,ip,n)
c
c     indirect addressing
c     indirect addressing on rhs
c
      integer n,ip(*)
      real a(*),b(*)
      do 550 i = 1,n
         a(i) = b(ip(i))
  550 continue
      return
```

```
      end
c%4.11
      subroutine s4113(a,b,ip,n)
c
c     indirect addressing
c     indirect addressing on rhs and lhs
c
      integer n,ip(*)
      real a(*),b(*)
      do 570 i = 1,n
         a(ip(i)) = b(ip(i))
  570 continue
      return
      end
c%4.11
      subroutine s4114(xx,yy,n)
c
c     indirect addressing
c     mixed variable and constant lb, ub, and stride
c@    n1,n2 are not defined.
c@    l is used before being defined
c
      dimension xx(100), yy(100), zz(100), l(100)
      xx(1) = 5.
      j = 5
      do 50 i=n1,n2
         k=l(i)
         xx(i)=yy(i)*zz(n2-k+1)
         k = k + j
   50 continue
      write (unit=6,fmt=11) j, xx(n)
   11 format (i6,e12.6)
      return
      end
c%4.11
      subroutine s4115(xx,yy,n)
c
c     indirect addressing
c     all variable - lb, ub, and stride
c@    will always vectorize since I is induction variable
c@    and XX appears only on left hand side.
c@    n1,n2,n3 are not defined.
c@    l is used before being defined
c
      dimension xx(100), yy(100), zz(100), l(100)
      xx(1) = 7.
      do 70 i=n1,n2,n3
         k=l(i)
         xx(i)=yy(2*i+1)*zz(k+i/n3+n1)
   70 continue
      write (unit=6,fmt=11) xx(n)
   11 format (i6,e12.6)
      return
      end
c%4.11
      subroutine s4116(n,d)
c
c     indirect addressing
c     This example test partial vectorization for the creation of
c     vector of indices. The best results are partial vecto
c     with the net effect being the loops being compiled as:
c
c     temp(1)=1
c     do i= 2,n
c     temp(i) = temp(i-1)*2 (actually temp(i-1)+temp(i-1))
c     enddo
c     do i= 1,n
c     d(temp(i)) =1
c     enddo
c
      dimension d(*)
      j =1
      do 40, i= 1,n
         j = j*2
         d(j) = 1
   40 continue
      return
      end
c%4.11
      subroutine s4117(a,b,c,n,m)
c
c     indirect addressing
c     seq function
c
      dimension a(1000),b(1000),c(1000)
      do 1200 i = 1,m
         a(i) = b(i) + c(i/2)
 1200 continue
      return
      end
```