A Dynamic Study of Vectorization in PFC

David Callahan
Ken Kennedy
Uli Kremer

CRPC-TR89015
September 1989

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A Dynamic Study of Vectorization in PFC

David Callahan Ken Kennedy Uli Kremer
david@tera.com ken@rice.edu kremer@rice.edu

Department of Computer Science
P.O. Box 1892
Rice University
Houston, Texas 77251

(713) 527-6077

Abstract

Today's supercomputers are mainly used to solve numerical problems. Understanding the characteristics of numerical programs is therefore essential for the design of supercomputers and their compilers. A run-time information gathering tool has been developed that allows (1) the study of program properties as they relate to vectorization and (2) the study of the efficiency of vector transformation used in vectorizing compilers. This paper describes the dynamic tool based on the PFC vectorizer and the results gained from the analysis of a testsuite of eight computationally intensive numerical programs.

1 Introduction

At Rice University empirical studies on the properties of Fortran programs have been performed and are still underway as they relate to parallel and vector architectures and the prospects for automatic detection of parallelism. These studies are based on a set of 'real world' programs that have been collected from industry and universities over the last nine months. RICEPS, the Rice Compiler Evaluation Program Suite, is a representative subset of these programs.

In order to facilitate dynamic studies of Fortran programs, a tool called PFC-SIM has been developed. PFC-SIM is a program-event-driven tracing facility which can be modified to perform a variety of dynamic statistic-gathering tasks.

The practical value of run-time information gathering tools is based on the assumption that information gained by executing a specific program with a specific input data set is also valid for other input data sets or other similar programs. This assumption is especially true for programs like oil reservoir codes where a specific computational model is specified by one part of the input data set while another part contains the actual data for the computation. Many input data sets share the first part but differ in the second.

Some studies have already been performed including studies on uniprocessor memory management [Por89], array subscript characteristics relevant for data dependence analysis, and vectorization potential and vector characteristics.

In this paper we present the results on the latter study, analyzing eight Fortran programs, some of which have been included in the program suite RiCEPS.

After a description of the vector version of PFC-SIM, called PFC-VECTSIM, in the next section we will briefly discuss the eight Fortran programs that were used in this study. The results of the study are given in section 4, followed by a discussion of the possibility of incorporating PFC-VECTSIM into an interactive programming environment.

2 PFC-VECTSIM

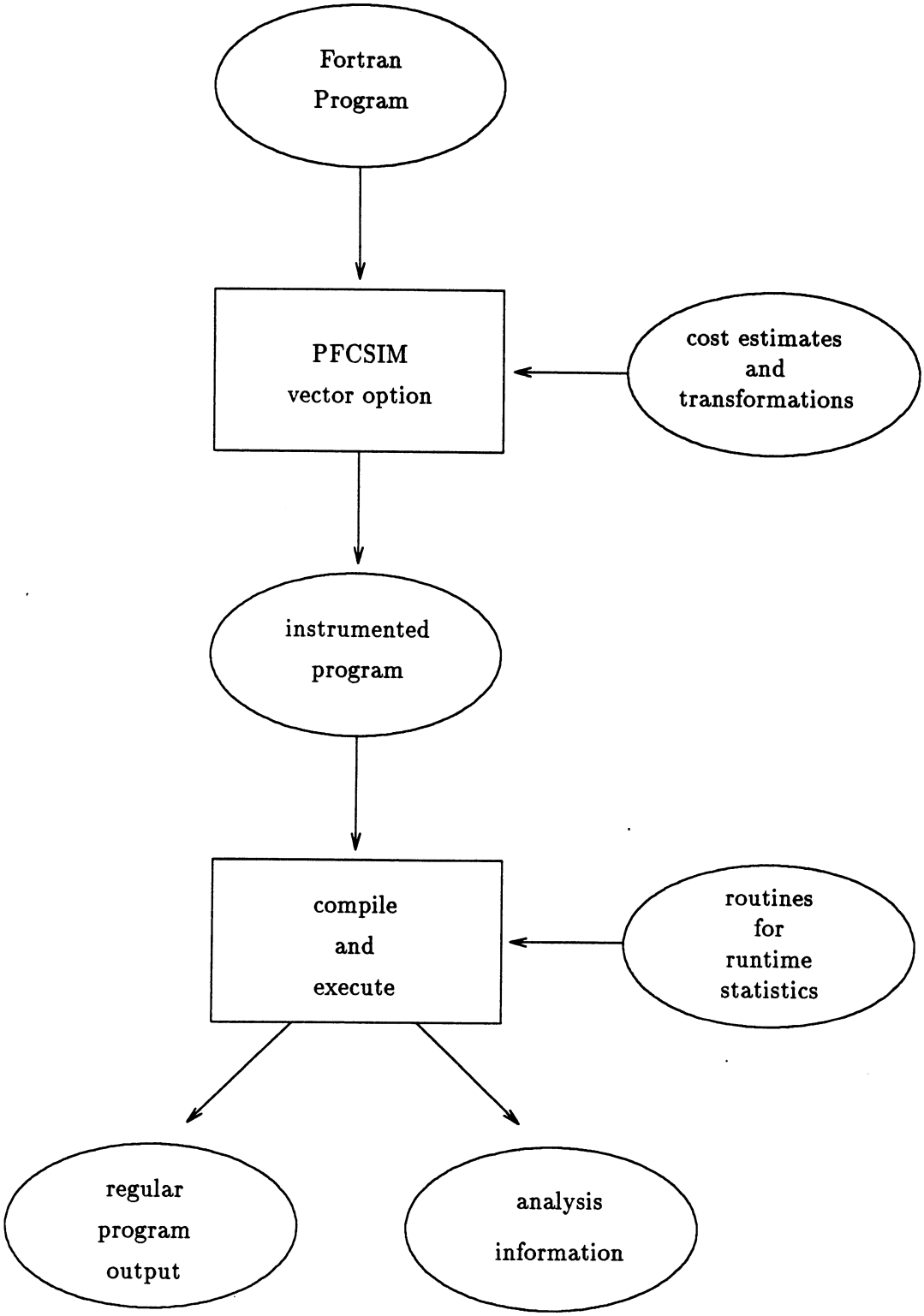
PFC-SIM is a tool to gather dynamic statistics of specific characteristics of Fortran77 programs. The input program is instrumented with analysis statements that compute the desired information during the execution of the program. This technique avoids long traces and therefore allows the analysis of programs that run for several hours or even days on an IBM3081D mainframe.

Two versions of PFC-SIM have been implemented so far, a version to gather information about the memory performance of the input program on a uniprocessor in terms of cache hits, misses and memory traffic (MEMORY OPTION[Por89]), and a version to determine the vectorizability and vector characteristics of the input program, using program restructuring transformations (VECTOR OPTION). Other versions to understand multiprocessor memory management and the parallelization potential of numerical programs are in preparation. In this paper we will concentrate on PFC-VECTSIM, the VECTOR OPTION version of PFC-SIM.

2.1 Structural Overview

Figure 1 shows the structure of PFC-VECTSIM. A modified version of the PFC VECTORIZER ([AK87], [All83]) is used that doesn't generate Fortran8X vector triplet notation [ANS86] but sequential loop nests instead, with innermost loops marked as vector loops. In addition, the PFC VECTORIZER is modified in such a way that the application of specific

FIGURE 1: Structure of PFC-VECTSIM



restructuring transformations can be disabled. This feature of PFC-VECTSIM allows a study of the profitability of transformations with respect to the overall increase of vectorizable operations and with respect to vector characteristics. The set of enabled transformations and a table of cost estimates for operations, instructions, and intrinsic functions are input to PFC-VECTSIM. The table is used to determine cost estimates for each basic block in the input program. The instrumented output program of PFC-VECTSIM contains calls to routines that gather analysis information during runtime. The instrumented program is compiled and executed. The output of the original program and the non-analysis output of the instrumented program are compared to ensure the correctness of the applied restructuring transformations. The analysis output includes the number of operations and instructions executed inside and outside of vector loops, vector lengths and vector stride distributions, and vector characteristics such as average vector-mask density.

In the following, the cost estimates for operations and instructions, the computed vector characteristics, and the sets of enabled transformations are described in more detail.

2.2 Operations and Instructions

PFC-VECTSIM supports three different counters for each source-level basic block in the input program. The counters contain a cost estimate of all (1) floating point operations, (2) integer operations, and (3) load/store and type conversion instructions, executed during one execution of the basic block. All operations involving variables of type `real`, `double precision`, `complex`, and `double complex` are considered to be floating point operations. The values of the counters are determined by a static analysis, that visits each statement in the basic block and determines its contribution to the overall cost of the basic block. Note that operations and instructions in index expressions of arrays are ignored in the computation of static counter values.

Cost estimates for each arithmetic and relational operation, and intrinsic function are input to PFC-VECTSIM. The values used in this study were determined by experiments conducted on an IBM3081D and are shown in Figure 2. The cost estimate for a type conversion instruction is set to 1, if it does not involve a variable of type `complex` or `double complex`, and 2 otherwise. The estimate for load/store instructions is the number of memory words referenced by the instruction. Each reference to a variable is counted as either a load

or a store, thus assuming a very naive register allocation technique.

The contribution of a basic block to the overall cost of executing the program is determined by multiplying the values of the static counters by the number of times the basic block is executed. The latter information is provided by *frequency counters* ([Knu71]), that are associated with each basic block (**$DYNCNT** variables in the example of Figure 3).

2.3 Vector Characteristics

PFC-VECTSIM does not generate vector triplet notation but sequential nested loops instead with the innermost loops marked as vector loops. Each execution of an innermost vector loop generates for each array reference in the loop a *vector*. The lengths of these vectors are set to the number of times the body of the innermost loop is executed.

In Figure 3 each execution of the innermost vector loop generates four vectors. Information about vectors is gathered dynamically by PFC-VECTSIM inserted calls to analysis routines. Statistics about the vector lengths distribution are obtained by calls to routine **$VECTL**. All vectors of an innermost vector loop have the same length. The number of vectors is passed as the second parameter to procedure **$VECTL**, and their lengths as the first.

The *stride* of a vector is the memory address increment used to move from one element of the vector to the next ([Sto87]). The stride of each vector represented by an array reference can be determined by analyzing the subscripts of the reference knowing the induction variable of the innermost vector loop and the dimensions of the array.

PFC-VECTSIM distinguishes three classes of strides, namely, *constant* strides, *indirect* strides, and *masked/random* strides. Indirect strides can occur when arrays are used in subscript expressions. Masked strides are generated for masked vectors, i.e. array references under a guard. If the stride cannot be shown to be constant, indirect ,or masked, a random stride is assumed. In the example of Figure 3, a stride of 0 is computed for the array reference A(I,K) , since in each iteration of the surrounding J-loop the same memory location is accessed. Calls to routine **$STRIDE** provide the means to compute vector stride distributions. The first parameter represents the class of the stride while the second gives the stride value. The stride value is only defined for the class of constant strides.

Vector mask densities are calculated as the ratio between the number of times a guard expression is evaluated and the number of times this evaluation yields the value *true*. All

FIGURE 2: Table of Operation Cost Estimates

operations	integer	real	double precision	complex	double complex
+, -	1	1	1	2	2
*	2	1	1	5	5
/	4	2	3	11	13
**	8	20	42	67	73
.gt.	2	1	1		
.eq.	2	1	1	7	7
sqrt		10	13	20	23
exp		8	18	28	33
log		13	18	32	37
log10		15	22		
sin		9	12	35	51
cos		8	14	35	51
tan		13	19		
asin		20	28		
acos		22	31		
atan		9	15		
sinh		11	30		
cosh		9	30		
tanh		11	28		
aint		1	1		
anint		2	2		
nint		3	3		
abs	1	1	1	7	8
mod	4	4	5		
sign	1	1	1		
dim	2	1	1		
max(2-arg)	2	1	1		
max(3-arg)	3	2	2		
aimag				1	
conjg				1	

array references in guarded statements are considered to be masked vectors.

2.4 Transformations

The set of restructuring transformations that can be applied during vectorization is input to PFC-VECTSIM. This feature of the tool allows us to gain insight into the profitability of transformations with respect to their effect on the overall vectorization and with respect to specific vector characteristics, e.g. vector lengths and vector strides.

The transformation sets used in this study are the following:

- **Set1**: Statement Reordering and Loop Distribution
- **Set2**: *Set1* + Loop Shifting
- **Set3**: *Set1* + If-Conversion and Scalar Expansion
- **Set4**: *Set3* + Loop Shifting and Loop Switching
- **Set5**: *Set4* + Array Renaming and Single Statement Reductions

Loop Shifting and Loop Switching are special forms of loop interchange as described in [AK84]. If there are no dependences at a level k, then the k loop can be placed as the innermost loop. This process is called *loop shifting*. *Loop switching* is loop interchange applied to only the innermost pair of loops in a π-block ([AK87]). The transformation *array renaming* is also known in the literature as *node splitting* ([KKP+81]). Single statement reductions are sum, product, and minimum/maximum reductions.

3 Test Suite

PFC-VECTSIM has been applied to a test suite of eight Fortran77 programs, some of which are members of the benchmark suite RiCEPS. A brief description of the eight programs and their major data structures is given below:

1. **LES**: (1240 lines and 24 program units)

   Large Eddy Simulation code; the program contains one to four-dimensional arrays of type *real*. The program uses two working space arrays of sizes $8192 \times 4 \times 4$ and 24576×4.

FIGURE 3: Matrix Multiply Example of $N \times N$-Matrices

```
      ...
      DO 1019 K = 1, N, 1
        $DYNCNT(9) = $DYNCNT(9) + 1
C$PFC Vector
        DO 1018 I = 1, N, 1
          $DYNCNT(7) = $DYNCNT(7) + 1
C$PFC Vector
          DO 1017 J = 1, N, 1
            $DYNCNT(5) = $DYNCNT(5) + 1
            $VECLNG = $VECLNG + 1
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
1017        CONTINUE
          $DYNCNT(6) = $DYNCNT(6) + 1
          CALL $STRIDE( 1, N )
          CALL $STRIDE( 1, 0 )
          CALL $STRIDE( 1, N )
          CALL $STRIDE( 1, N )
          CALL $VECTL( $VECLNG, 4 )
          $VECLNG = 0
1018      CONTINUE
        $DYNCNT(8) = $DYNCNT(8) + 1
1019    CONTINUE
      ...
```

Statements in **typewriter style** have been inserted by PFC-SIM VECTOR OPTION

2. **LINPACKD:** (796 lines and 11 program units)

   Double precision LINPACK test routines. The computations are performed for arrays of size 100×100 of type *double precision.*

3. **NASKER:** (1183 lines and 19 program units)

   NAS kernel benchmark program; the major data structures are arrays of one to four dimensions that share a working space common block of 360000 *reals.*

4. **ONEDIM:** (1016 lines and 16 program units)

   Eigenfunction and Eigenenergies calculation of the time-independent Schrodinger equation for a one-dimensional potential. The major data structures are two-dimensional arrays of size 150×150 of type *double precision.*

5. **SHAL64:** (534 lines and 9 program units)

   Benchmark weather prediction program using a simple atmospheric dynamic model based on shallow-water equations. Two-dimensional arrays of size 65×65 of type *real* are the basic data structures.

6. **SHEAR:** (915 lines and 15 program units)

   Three-dimensional turbulence fluid dynamics simulation based on spectral techniques. The computations involve three-dimensional arrays of size $18 \times 17 \times 17$ of type *real* that are aliased to a working space array of size $18 \times 17 \times 238$.

7. **SIMP1:** (1312 lines and 8 program units)

   Two-dimensional Lagrangian hydrodynamics code with heat diffusion (Standard LLNL benchmark code). The major data structures are one-dimensional arrays of size 12 and 203, and two dimensional arrays of size 203×183. The arrays are of type *real.*

8. **VORTEX:** (709 lines and 20 program units)

   Particle dynamics code simulating the dynamics of a one-dimensional vortex sheet by means of discrete vortices. The computation involves one-dimensional arrays of type *real* with sizes 48 and 1000.

4 Results

The results of the study with respect to the number of vectorized operations and instructions are given in Figure 4. Neither interprocedural side effect analysis nor interprocedural constant propagation were performed. A comprehensive summary of the vector characteristics can be found in the appendix.

In Figure 4, the program names are followed by a list of numbers in parentheses. The numbers in the list represent transformation sets that generate the same instrumented output program. The profitability of transformations can be determined using the set of instrumented programs generated and their corresponding vector/scalar ratios. For instance, in the case of LINPACKD, the transformation set 1 generates a program that achieves a floating point vector ratio of 92.3%. Since transformation sets 1 and 2 produce the same program loop shifting alone is not profitable after statement reordering and loop distribution. The increase of the floating point vector ratio by 4.1% to 96.4% is only due to if-conversion and scalar expansion, because the additional transformations of transformation set 3, namely loop shifting and switching, could not improve the vector ratio since the output programs of transformation sets 3 and 4 are the same. Therefore loop shifting and loop shifting are not profitable at all for vectorizing LINPACKD. The final improvement of the floating point vector ratio by 1.4% to 97.8% is obtained by applying single statement reductions.

In general the PFC VECTORIZER achieved good results for all programs except SHEAR. For SHEAR the vectorizer failed to vectorize two loop nests that contribute to more than 40% of the overall number of executed floating point operations, even when interprocedural side effect analysis and constant propagation had been switched on. The interprocedural constant propagation was not able to propagate three constants that were stored in an array N and referenced in the program as N(1), N(2), AND N(3), respectively. These three constants were crucial for the precision of the dependence analyzer which - not knowing the constant values - had to make conservative assumptions about the existence of dependences in the two loop nests. After substituting three 'new' scalar variables for the array N, PFC vectorized the two computationally intensive loops using interprocedural constants and side effect information.

The high vectorization ratio and the dominance of unity strides due to applying the first transformation set for programs LES, SHAL64 and SIMP1 seems to indicate that these

FIGURE 4: Table of Operations and Instructions

programs[1]	integer operations			floating point operations[2]			load, store, type conversion[2]		
	scalar	vector	ratio[3]	scalar	vector	ratio[3]	scalar	vector	ratio[3]
LES(1)[4]	2218294	26214924	92.2 %	4.6	3158	99.9 %	16.2	9341	99.8 %
LINPACKD(1,2)	2868768	0	0 %	1.5	18	92.3 %	9.8	72.8	88.1 %
LINPACKD(3,4)	2866194	5148	0.2 %	0.7	18.8	96.4 %	0.8	74.7	90.4 %
LINPACKD(5)	2866194	5148	0.2 %	0.4	19.1	97.8 %	6.3	76.3	92.4 %
NASKER(1,2)	471608	0	0 %	1152.4	1243.1	51.9 %	1579.9	1782.5	53.0 %
NASKER(3,4)[5]	2117806	192	0 %	11.9	2383.6	99.5 %	28.7	3335.5	99.1 %
NASKER(5)	2117796	202	0 %	6.9	2388.6	99.7 %	13.7	3350.5	99.6 %
ONEDIM(1)	738166	154	0 %	76.8	11.5	13.0 %	281.8	42.3	13.1 %
ONEDIM(2)	738166	154	0 %	56.5	31.7	36.0 %	200.8	123.3	38.0 %
ONEDIM(3)	1996710	2487154	55.5 %	36.5	51.8	58.7 %	121.5	206.0	62.9 %
ONEDIM(4)[6]	2.0	2.5	55.5 %	16.2	72.0	81.6 %	40.5	287.0	87.6 %
ONEDIM(5)	1996710	2487154	55.5 %	9.6	78.9	89.2 %	13.8	313.6	95.8 %
SHAL64(1-5)	132011	8450	6.0 %	0.5	3244.1	99.9 %	2.0	4512.8	99.9 %
SHEAR(1,2)	5629575	136	0 %	331.5	155.0	31.9 %	508.5	362.5	41.6 %
SHEAR(3,4)[4]	5629223	488	0.01 %	310.3	176.2	36.2 %	466.0	404.9	46.5 %
SHEAR(5)	5629223	488	0.01 %	307.5	179.0	36.8 %	457.7	413.3	47.5 %
SHEAR(5*)[7]	5629223	488	0.01 %	90.2	396.4	81.5 %	227.5	643.4	73.9 %
SIMP1(1,2)	484624	0	0 %	174.7	8754.7	98.0 %	363.1	8651.5	96.0 %
SIMP1(3,4)	213040	271584	56.0 %	92.1	8837.4	98.9 %	145.6	8869.0	98.4 %
SIMP1(5)	212610	272014	56.1 %	86.7	8842.7	99.0 %	129.5	8885.1	98.6 %
VORTEX(1,2)	28499	10500	26.9 %	1367.9	0.8	0.06 %	3255.2	2.3	0.08 %
VORTEX(3)	29499	10500	26.2 %	189.3	1179.4	86.2 %	568.1	2689.3	79.9 %
VORTEX(4)[6]	29499	10500	26.2 %	1.3	1367.4	99.9 %	4.3	3253.2	99.8 %
VORTEX(5)	29499	10500	26.2 %	0.8	1367.9	99.9 %	2.6	3254.9	99.9 %

[1] applied transformation sets are listed in parentheses
[2] in units of 10^6
[3] ratio := (vector / (scalar + vector)) * 100
[4] numbers for transformation sets 1 - 5 differ only slightly
[5] if-conversion didn't improve vectorization
[6] loop switching didn't improve vectorization
[7] interprocedural constant propagation and side effect analysis

programs have been written for vector machines, i.e. many vectorization transformations have been performed manually.

If-conversion was very successful in the case of VORTEX where nearly every loop nest contains control flow, but failed to significantly improve vectorization for all other programs. Loop shifting also has a mixed review. It was useful for ONEDIM and VORTEX but had no effect in the cases of LINPACKD, NASKER, SHEAR, and the manually vectorized programs. For all programs except ONEDIM, applying single statement reductions like sum, product and minimum/maximum reductions lead to only slight improvements of the vector ratios although they were counted as full vector operations. Loop switching, array renaming, and product reductions did not improve vectorization of any program in the test suite. Note that for some programs, namely LINPACKD, NASKER, ONEDIM, and VORTEX the number of integer operation increases after if-conversion and scalar expansion has been applied. This increase is due to the fact that if-conversion introduces boolean guards to eliminate goto or if-then-else statements. For instance, in the latter case the else-statement can be substituted by an if-statement that contains the negation of the condition for the true-branch.

Vector characteristics like vector lengths and vector strides depend on the choice of the innermost loop in a vector loop nest. Therefore loop interchange can change the number of vectors generated by a vector loop nest, and their vector characteristics. PFC-VECTSIM does not alter the order of loops in a loop nest of the input program unless loop interchange leads to more inner loops that carry no dependence and therefore can be vectorized. In particular, loop shifting is not applied to maximize vector lengths or to minimize vector strides.

The vector characteristics of the different programs varied significantly. While SHAL64 had only two distinct vector lengths 64 and 65, SIMP1 generated vectors of over twenty different lengths in the range of 1 to 203. LINPACKD, ONEDIM, and VORTEX showed vector length distributions typical for triangular matrix computations. Vector strides of 0 and 1 only dominated the stride distributions of LES, LINPACKD, SHAL64, SIMP1. Stride distributions of programs NASKER, ONEDIM and SHEAR showed constant strides that indicate computations involving row-vectors as well as column-vectors. Vectors with indirect strides only occurred in the case of SIMP1 and there were no vectors with random strides.

For a closer study of the vector characteristics of each program we refer the reader to the

summary table and diagrams in the appendix.

5 Conclusion and Future Work

The study of eight computationally intensive numerical programs shows that automatic vectorization techniques are able to vectorize most of the computations involved in executing these programs. For some programs a high vectorization ratio and a dominance of unity strides are achieved without performing vectorization transformations such as scalar expansion, if-conversion, or loop interchange. This seems to indicate that in these cases programmers have adopted a 'vector' programming style, i.e. most vectorization transformations have been performed manually by the programmer.

Run-time information gathering tools can be useful for hardware and compiler designers if applied to programs that are considered to be typical applications for the envisioned architecture or compiler. Tools like PFC-SIM can also help optimize a specific program or estimate the runtime improvement resulting from executing the program on a vector machine instead of a sequential machine as in [PIJJ88]. Embedding PFC-SIM in an interactive environment would allow the user to identify performance bottle necks by mapping the generated information back to the source program. For the memory option of PFC-SIM such an interactive visualization tool already exists ([Por89]). A similar tool for the vector version could easily be implemented. This tool would give the user the ability to perform vector optimizations ([Wol87]), or to identify computationally intensive loops that have not been vectorized automatically. The latter feature seems to be extremely useful. For instance in SHEAR, two loop nests contribute to more than 40% of the performed floating point operations. Due to conservative assumptions in the dependence analyzer the PFC VECTORIZER fails to vectorize these loops automatically. The user is able to concentrate his effort to vectorize a program on computationally intensive loops. After analyzing the problem the user can either change the program in such a way that the vectorizer is able to detect the vector loops or he can insert compiler directives that provide assertions about the values of specific variables and thereby enable vectorization.

The dynamic nature of PFC-VECTSIM allows an insight into the behavior of programs that cannot be gained by any static analysis. Due to the representation of vectors by vector

loop nests and the insertion of instrumentation code the output program of PFC-VECTSIM is about three to four times the size of the input program. However, since calls to statistic gathering routines are not inserted into innermost loops the increase in execution time is not very significant. Tools to allow dynamic studies on multiprocessor memory management and parallelization potential are in the planning stage.

Acknowledgement

We would like to thank Allan Porterfield for his support and helpful comments during the development of PFC-VECTSIM.

References

[AK84] John R. Allen and Ken Kennedy. Automatic Loop Interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, pages 233–246, June 1984.

[AK87] John R. Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[All83] John R. Allen. Dependence Analysis for Subscripted Variables and its Application to Program Transformations. Technical report, Dept. of Mathematical Science, Rice University, April 1983. Ph.D. Dissertation.

[ANS86] American National Standards Institute X3J3:Fortran 8X Version 96. January 1986.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Conference Record of the Eigth ACM Symposium on the Principles of Programming Languages*, pages 207–218, January 1981.

[Knu71] Donald E. Knuth. An Empirical Study of Fortran Programs. In *Software Practice and Experience*, volume 1, pages 105–133, 1971.

[PIJJ88] D.J. Pease, R. Inamdar, A. Joshi, and S. Jejurikar. Predicting the Performance of a Scalar Program Converted to Execute on a Vector Processor. In *Proceedings of the 3rd ACM SIGSoft/SIGPlan Conference on Parallel Processing*, pages 355–361, August 1988.

[Por89] Allan Porterfield. Software Methods for Improvement of Cache Performance on Supercomputer Applications. Technical Report TR88-93, Rice University, May 1989. Ph.D. Dissertation.

[Sto87] Harold S. Stone. High-Performance Computer Architecture. Addison-Wesley, October 1987.

[Wol87] Michael Wolfe. Vector Optimization vs. Vectorization. In *Proceedings of the 1987 International Conference on Supercomputing (Athens)*, June 1987.

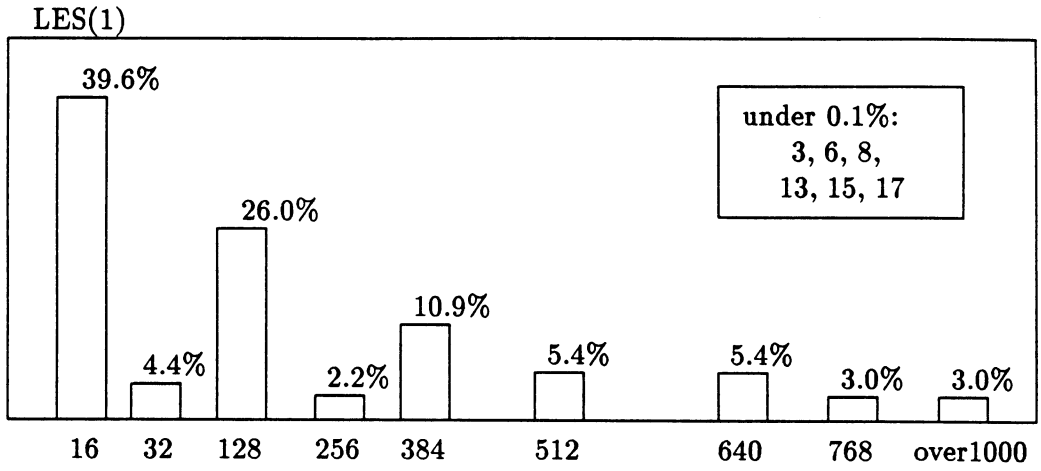
Appendix

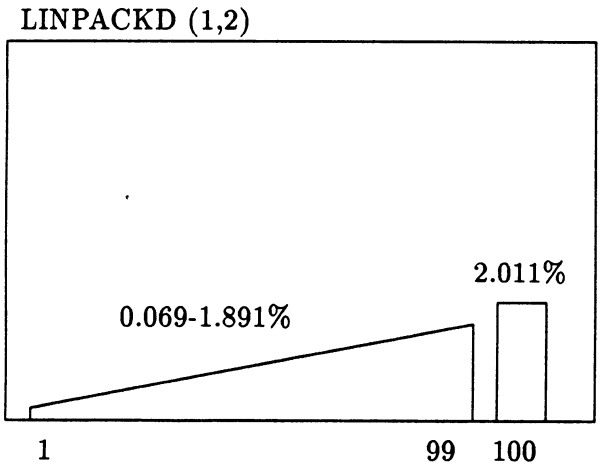
A - Vector Characteristics (Summary)

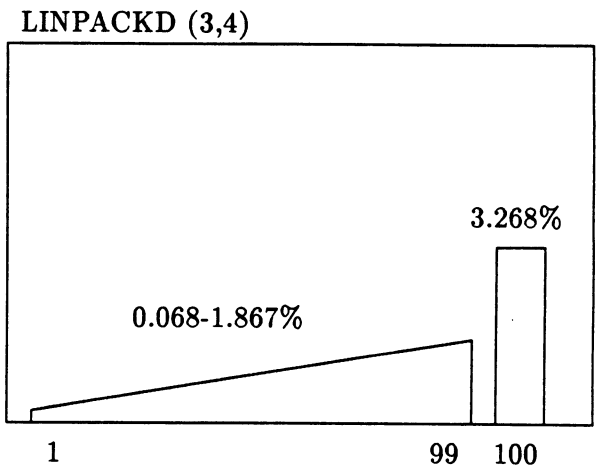
programs[1]	vectors	masked		lengths	lengths ($\leq$ 1000)		strides[2]	
	overall #	#	density	#(> 1000)	average	deviation[3]	average	deviation[3]
LES(1)	28027890	0		841628	179.48	214.99	0.91	0.29
LINPACKD(1,2)	415037			0	66.20	24.25	1	0.02
LINPACKD(3,4)	420515	0		0	66.64	24.39	1	0.02
LINPACKD(5)	423217	0		0	66.85	24.46	1	0.02
NASKER(1,2)	14454024			2840	113.04	117.43	421.10	370.42
NASKER(3,4)	25715300	0		2842	99.58	105.46	293.11	359.10
NASKER(5)	25720375	0		2842	99.76	106.19	293.05	359.09
ONEDIM(1)	233143	0		0	65.19	42.70	2.05	12.48
ONEDIM(2)	503143	0		0	110.70	51.31	28.0	57.57
ONEDIM(3)	823808	174000	0.68	0	83.97	65.25	1.54	7.57
ONEDIM(4)	1093808	174000	0.68	0	100.27	62.43	15.89	44.62
ONEDIM(5)	1182608	174000	0.68	0	98.37	61.62	14.58	42.82
SHAL64(1-5)	57454956	0		0	64.00	0	1.35	4.70
SHEAR(1,2)	16979053			8810	15.75	2.87	126.35	145.48
SHEAR(3,4)	22035520	2	0.28	8810	12.83	5.91	139.38	148.36
SHEAR(5)	22500028	2	0.28	8810	12.62	6.02	142.82	148.72
SIMP1(1,2)	90550096			0	84.62	49.12	20.53	59.70
SIMP1(3,4)	92871044	556000	0	0	84.12	49.19	20.34	59.46
SIMP1(5)	92931914	556000	0	0	84.12	49.20	20.35	59.45
VORTEX(1,2)	22512			0	50.34	94.53	1	0
VORTEX(3)	6855537	2986511	0.99	0	172.00	175.88	0.86	0.35
VORTEX(4)	9003070	2981022	0.99	0	172.73	176.04	0.82	0.38
VORTEX(5)	9007074	2981022	0.99	0	172.71	176.03	0.82	0.38

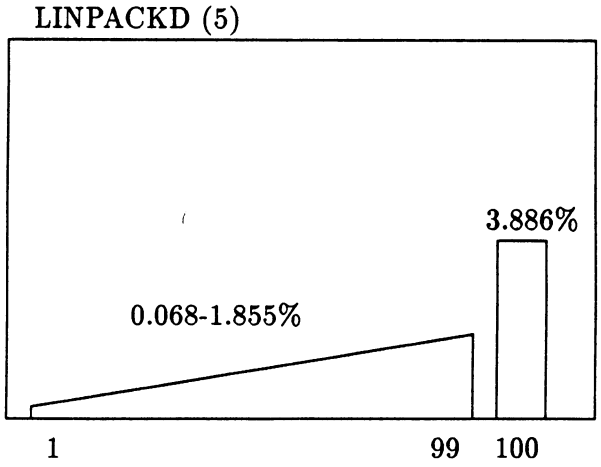
[1] applied transformation sets are listed in parentheses
[2] masked vectors don't contribute to the listed numbers
[3] standard deviation

B - Vector Length Distributions



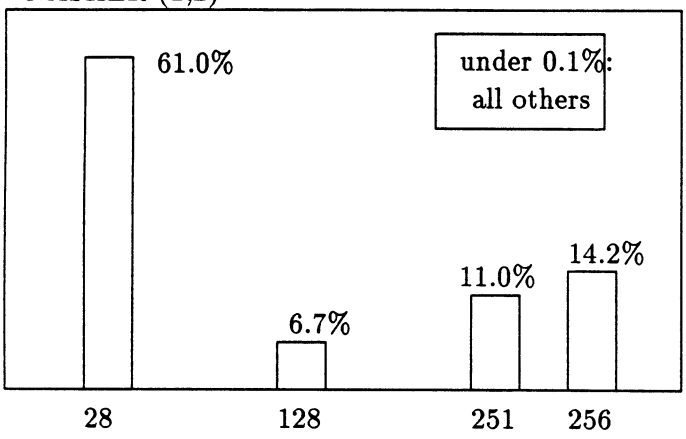




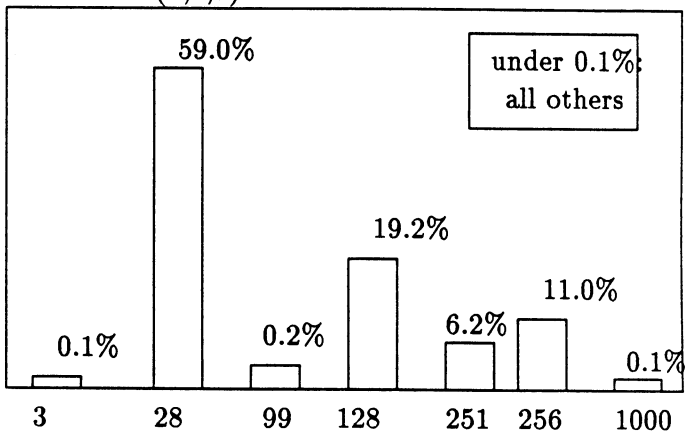


B - Vector Length Distributions (cont.)

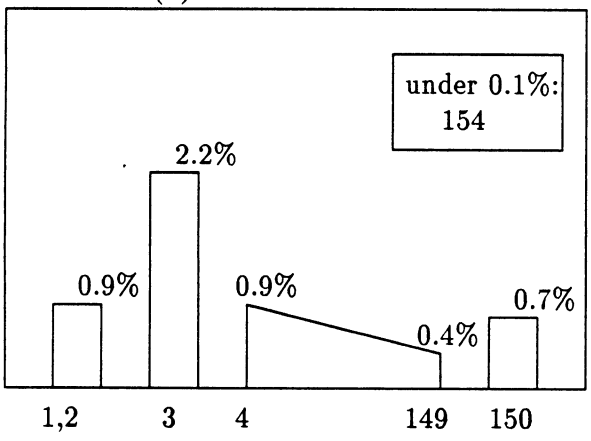
NASKER (1,2)



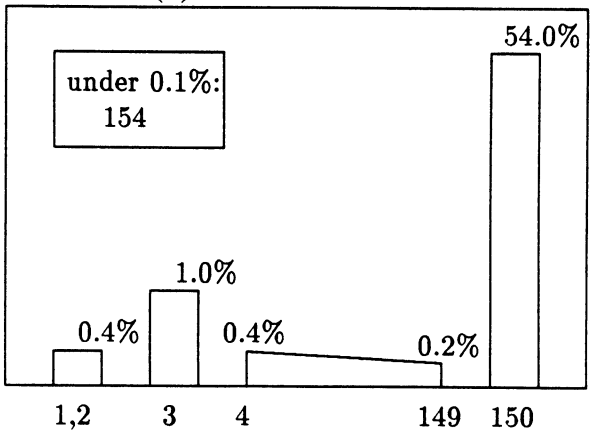
NASKER (3,4,5)



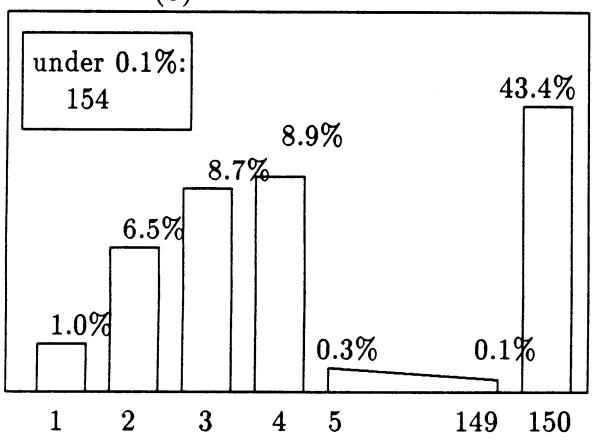
ONEDIM (1)



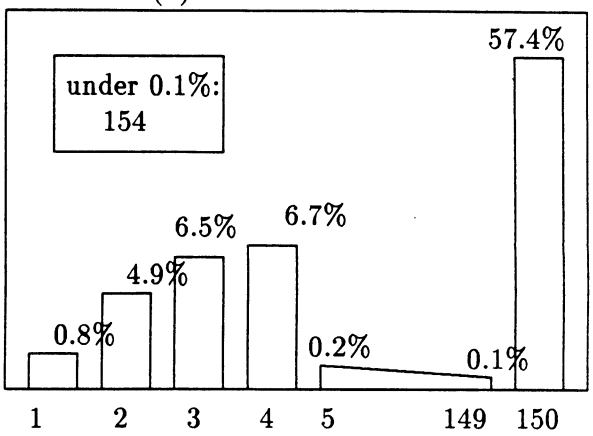
ONEDIM (2)



ONEDIM (3)

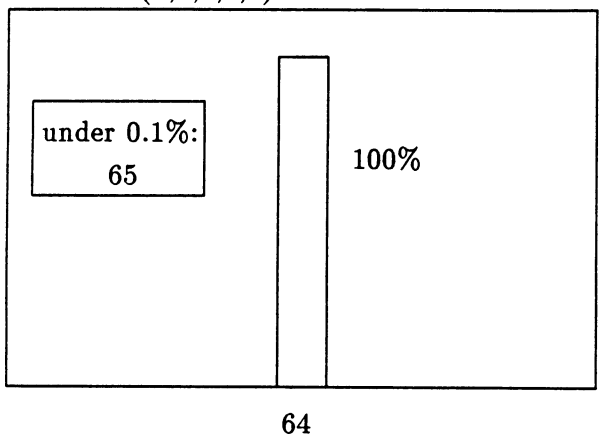


ONEDIM (4)

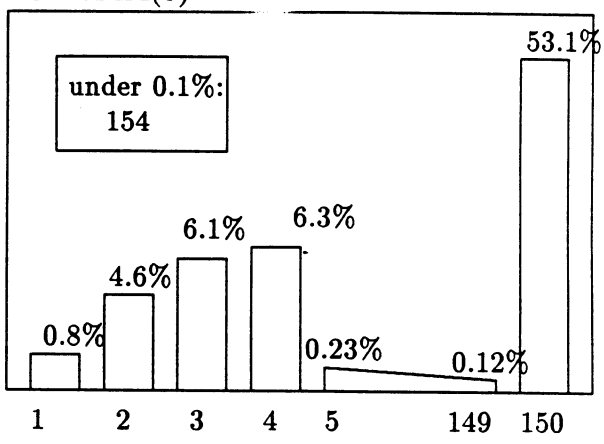


B - Vector Length Distributions (cont.)

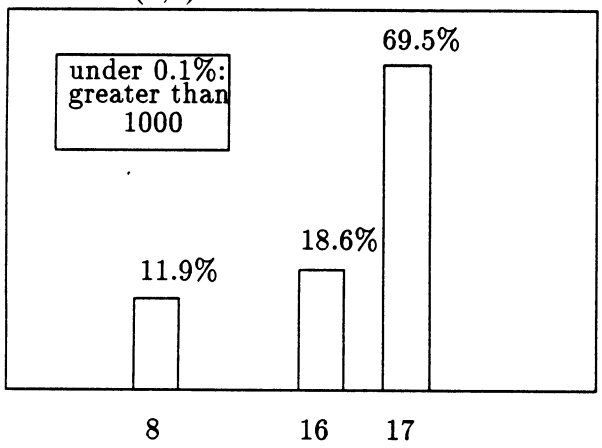
SHAL64 (1,2,3,4,5)



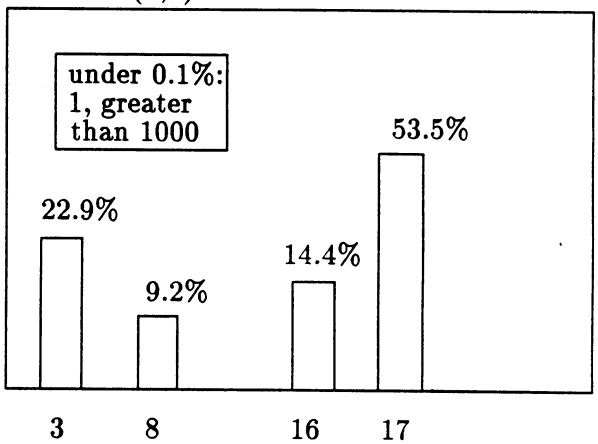
ONEDIM(5)



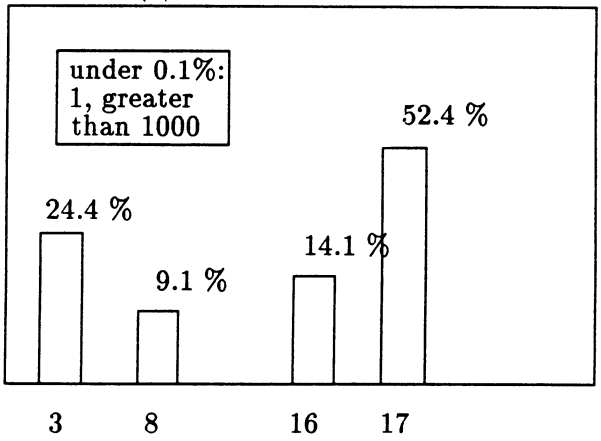
SHEAR (1,2)



SHEAR (3,4)

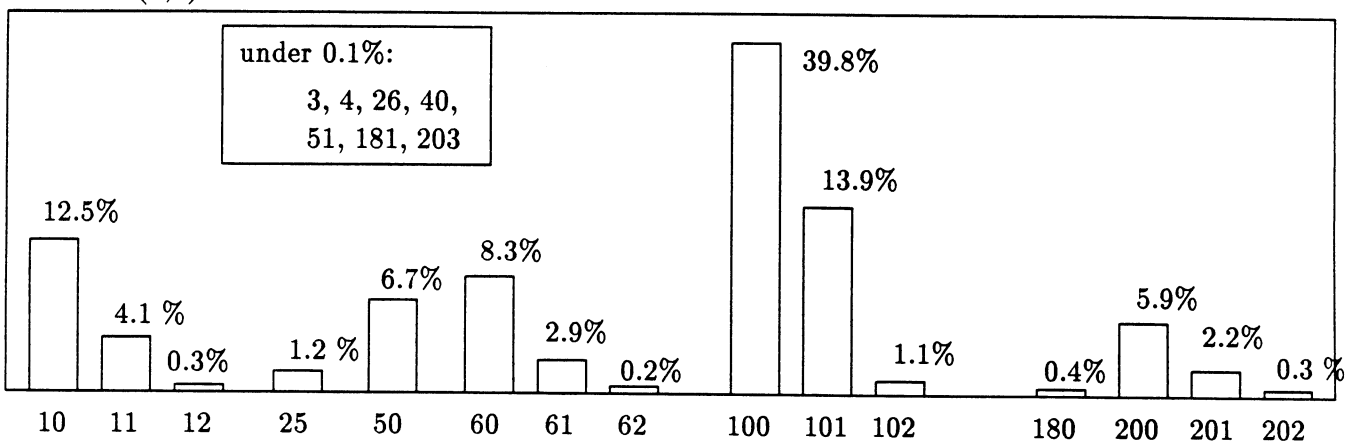


SHEAR (5)

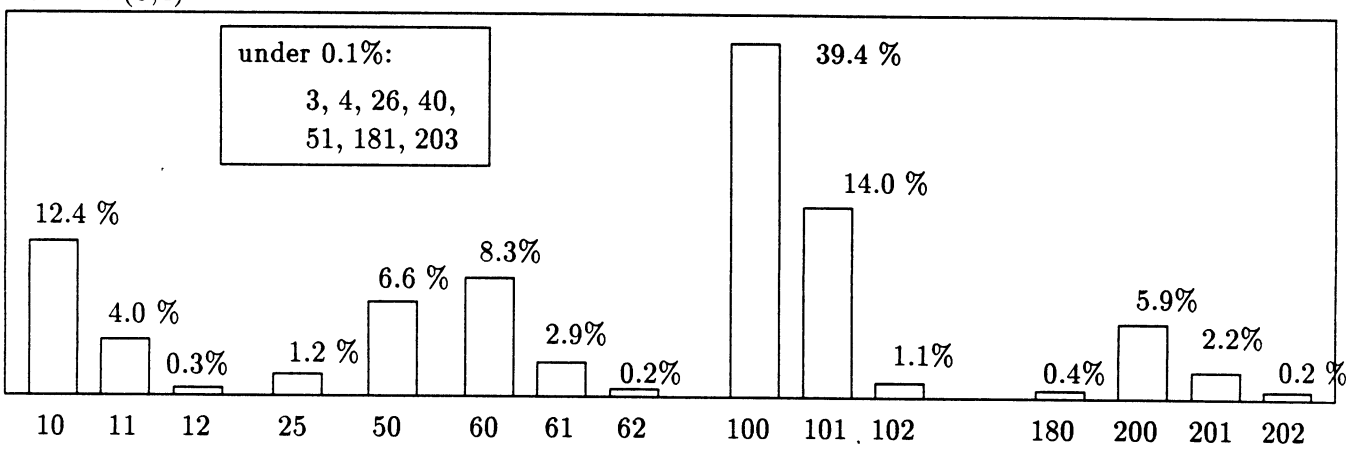


B - Vector Length Distributions (cont.)

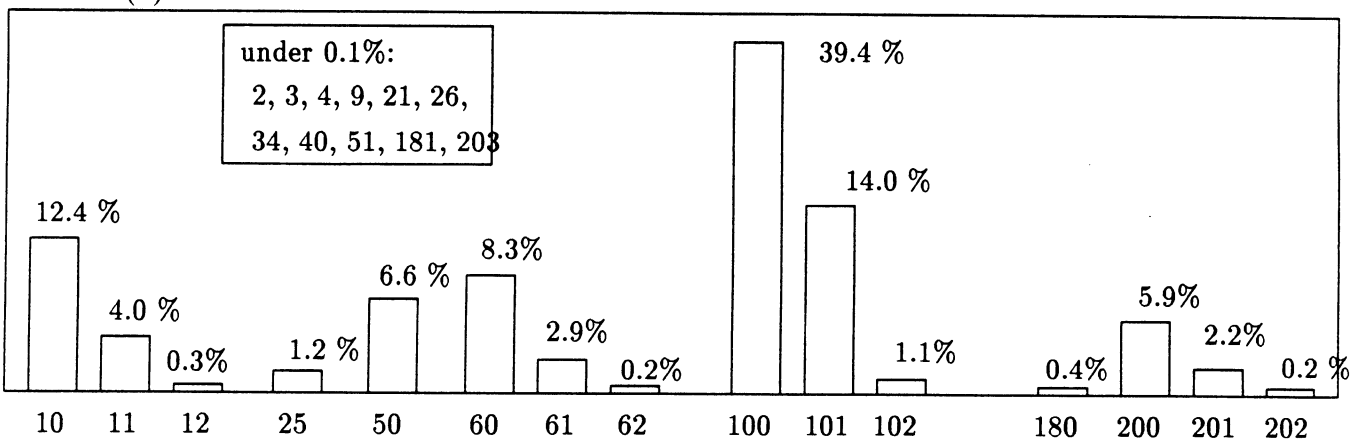
SIMP1 (1,2)



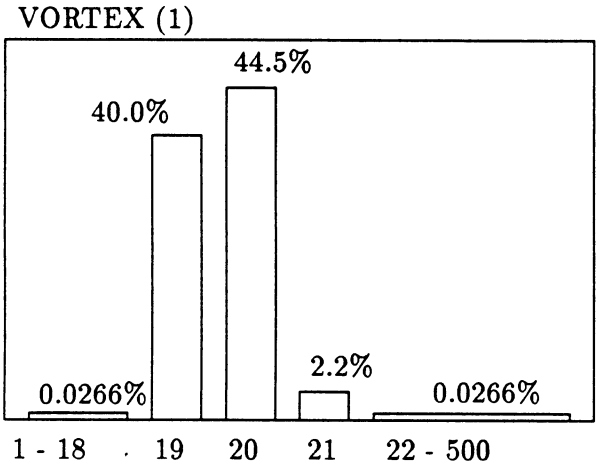
SIMP1 (3,4)

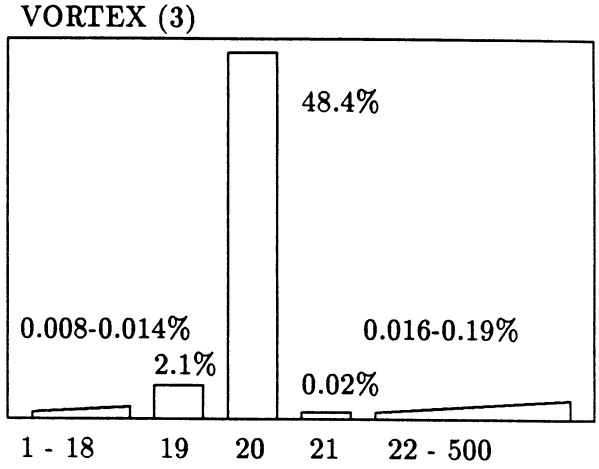


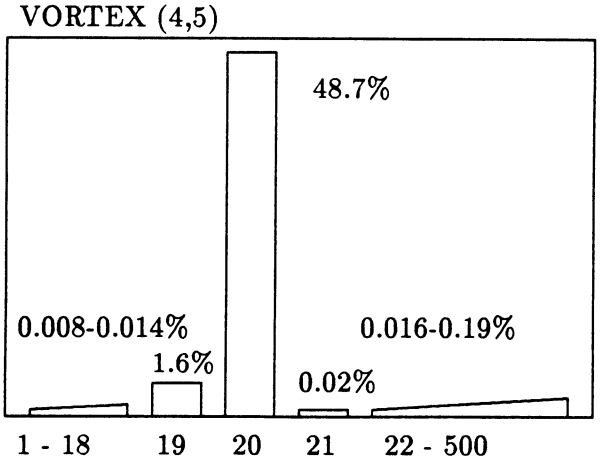
SIMP1 (5)



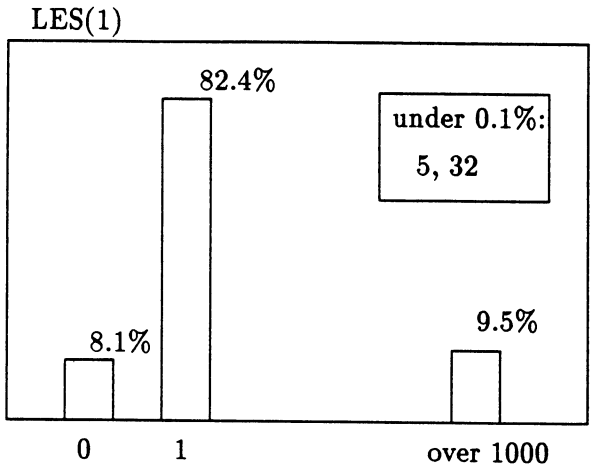
B - Vector Length Distributions (cont.)

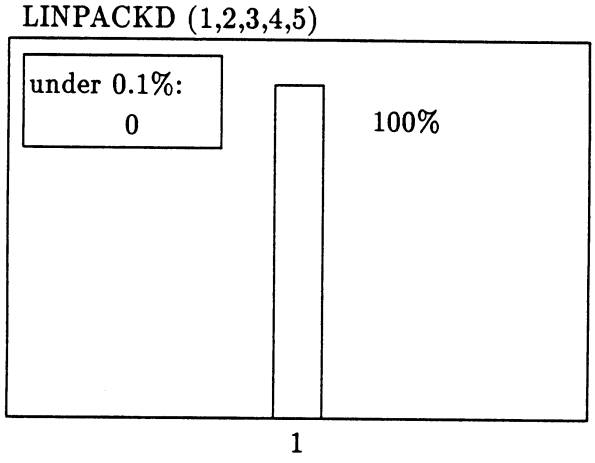


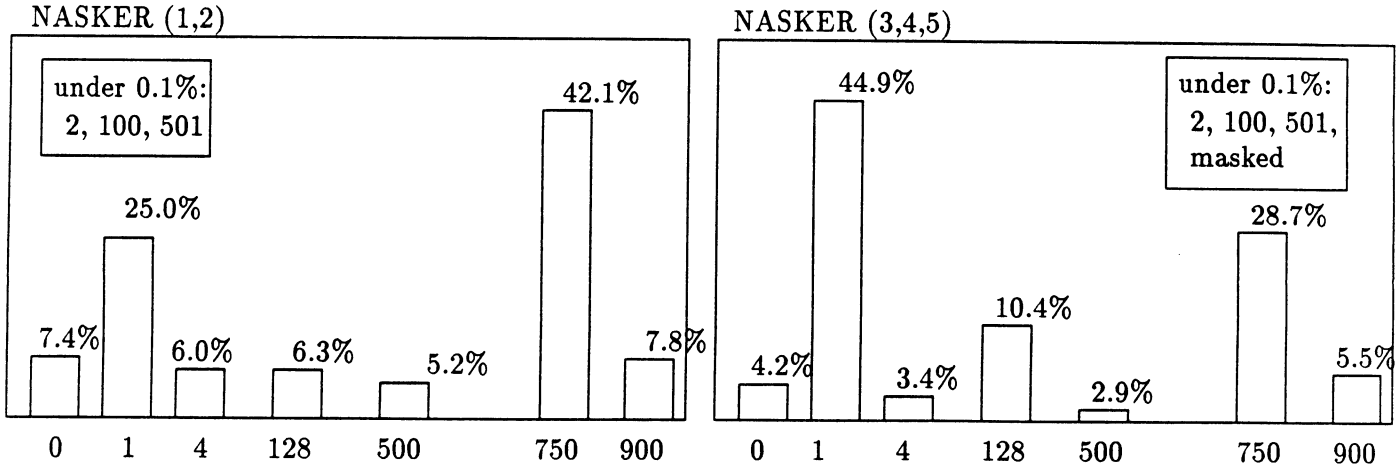




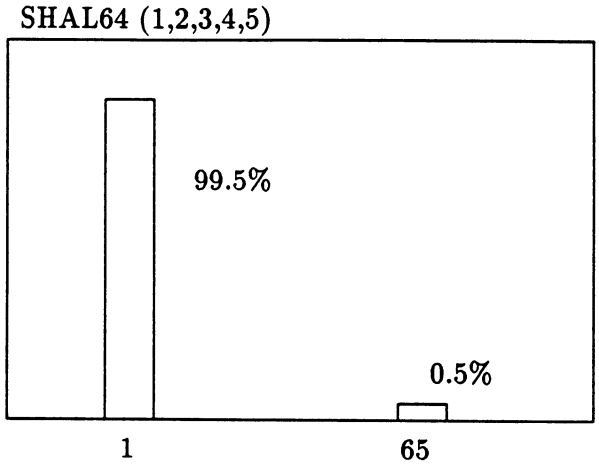
C - Vector Stride Distributions

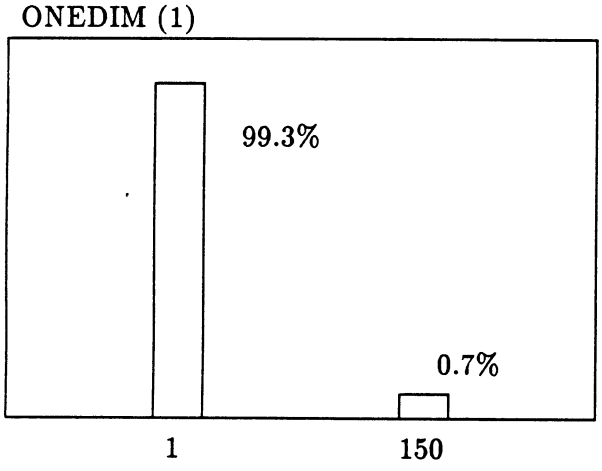


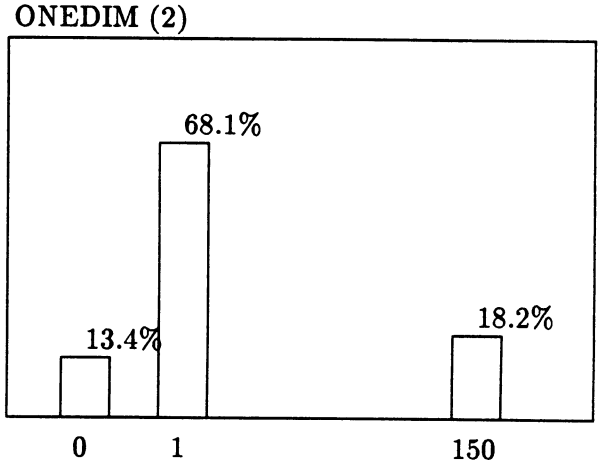


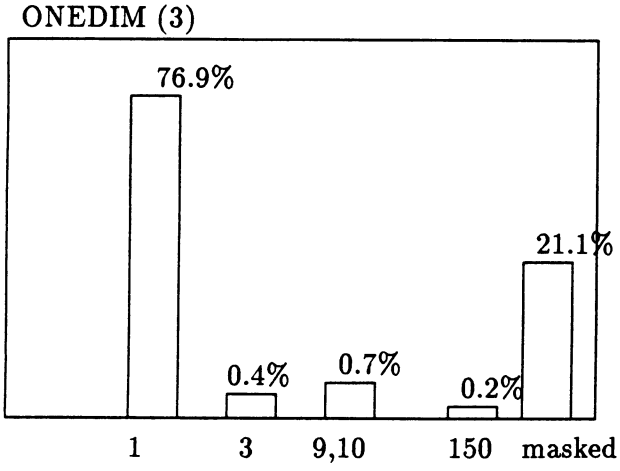


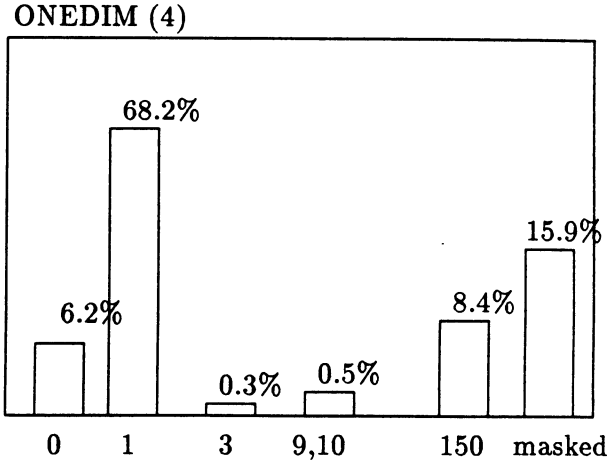
C - Vector Stride Distributions (cont.)



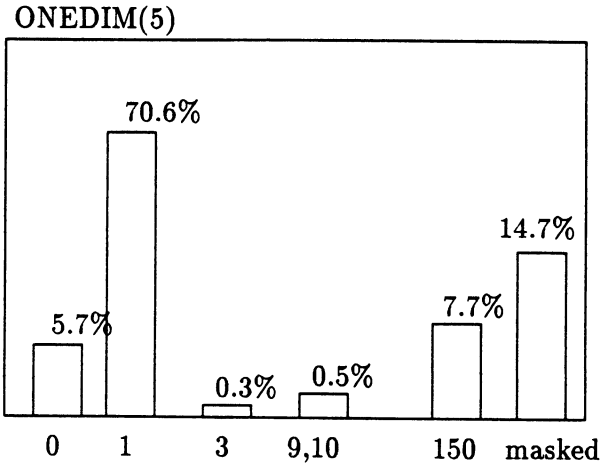


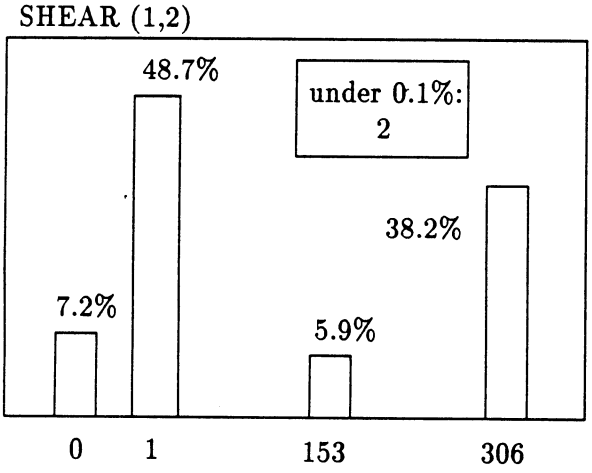


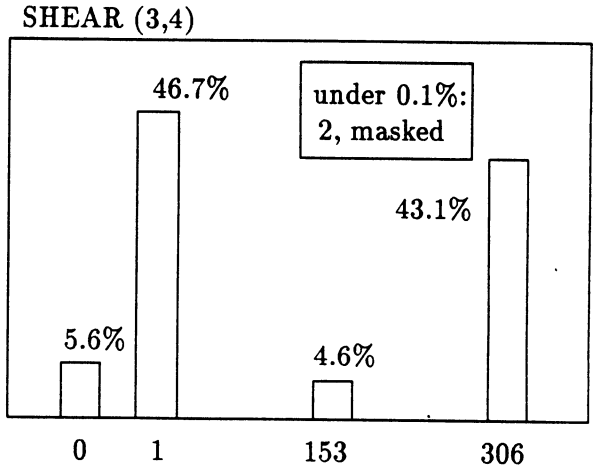


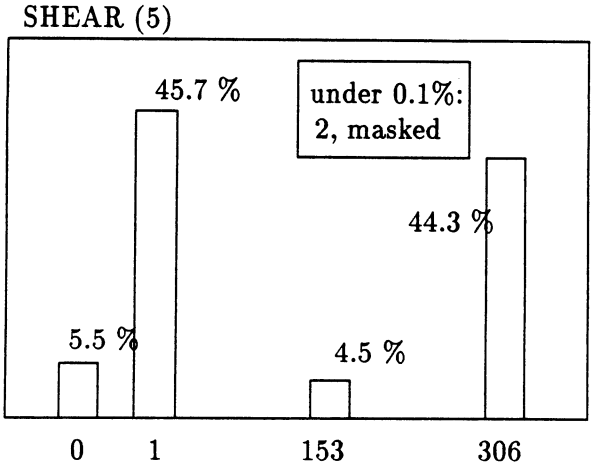


C - Vector Stride Distributions (cont.)









C - Vector Stride Distributions (cont.)