Vectorizing C Compilers: How Good Are They?

Lauren L. Smith
Visiting Member of the Research Staff
Supercomputing Research Center
17100 Science Drive
Bowie, MD., 20715-4300

Abstract

The programming language C is becoming more and more popular among users of high performance vector computer architectures. With this popularity of C, it becomes more critical to have a good optimizing/vectorizing C compiler. This paper describes a study of four such vectorizing C compilers, with emphasis on the automatic vectorization ability of each compiler. This study is similar to the Fortran study that was described in [CDL88] and in fact, one facet of this study is a C version of the same kernels. Three suites of C loop kernels have been developed to determine the strengths and weaknesses of vectorizing compilers. The Convex cc compiler, the Convex Application Compiler, the Cray 2 scc compiler, and the Cray YMP scc compiler have been tested against these suites. The paper gives the results for each suite, with identification of problem areas for each compiler.

1. Introduction

Programmers everywhere are obtaining easy access to high-performance workstations that are running UNIX. With this trend towards UNIX, more programmers are using C for scientific codes on vector supercomputers. Therefore, it becomes necessary to look at the capabilities and performance of C compilers for vector architectures.

Many of the current vectorizing C compilers use the same vectorizing techniques that were developed for Fortran. Are these techniques adequate? This paper will try to investigate the capabilities of current vectorizing C compilers and determine if additional techniques are needed. Key features of C, such as pointers and dynamically allocated memory objects will be examined with respect to vectorization on current vector architectures.

A multi-faceted approach has been undertaken to try to understand the capabilities of two vendors' C compilers. The vectorizing C compilers used for this study are Version 4.1 of the Convex C2 Vectorizing C Compiler (**cc**) [Con91a] [Con91b], Version 1.0 of the Convex Application C Compiler (**ac**) [Con91c], and Release 3.0.0 of the Cray Standard C Compiler (**scc**) on the Cray 2 and Cray Y-MP [Cra90]. Since this study is testing the compilers' vectorization abilities, no user directives or special compilation flags are used. Many of these kernels can be vectorized if the user uses such directives or flags, but this involves user analysis of their code which violates the spirit of testing automatically vectorizing C compilers.

Section 2 discusses the C version of the Argonne test suite for vectorizing compilers [CDL88]. The ability of the Convex and Cray C compilers to vectorize the suite is contrasted with the Fortran compilers and with each other.

Section 3 describes a continuation of the Argonne test suite study, but with purely unique C features and constructs. Comparisons are made between the vectorizing capabilities of the Cray and Convex C compilers. Some observations are also made on certain C language features that impact the vectorizing capability of a compiler.

Section 4 discusses the results of looking at a suite of C kernels abstracted from scientific C applications. Again, the ability of the Cray and Convex C compilers to vectorize these application kernels is contrasted and some comments are made on the actual use of certain C features.

It should be mentioned that some compiler terminology will be used to describe the abilities of the compilers. The reader might wish to look at [ASU96], [Ban88], [Pol88] or [Wol82] for definitions and a better understanding of some of the terms.

2. C version of the Argonne test suite

A suite of Fortran loop kernels was collected at Argonne National Laboratory to test the effectiveness of automatic vectorizing Fortran compilers [CDL88]. The loops were written by writers of vectorizing compilers, and test for specific vectorization features. Some results

for several vector architectures have been reported in [CDL88] and [Nob89].

The Argonne test suite was translated from Fortran to C adhering to a Fortran style. Some of the kernels explicitly test certain Fortran constructs, and were not translated, so a total of 91 out of 100 loops were successfully translated.. This suite of 91 kernels was then compiled using the vectorizing options of both the Convex and Cray compilers. As a point of comparison, the suite of 91 kernels was also compiled using the vectorizing Fortran compilers on both architectures. The Fortran compiler used on the Cray 2 was Version 4.0.3 of **cft77** along with Version 3.0 of **fpp.** The Fortran compiler used on the Convex C2 was Version 6.1 of **fc.**

The first thing discovered was that the Cray and Convex **cc** compilers would not fully vectorize loops where the arrays are passed in as arguments to the kernel. When arrays are passed to functions in C, the pointer to the array is passed. The only compiler that does the necessary interprocedural analysis for this problem is Convex's Application Compiler. Figure 1 demonstrates a kernel where this problem arises. Figure 2 shows how changing the arrays to global variables and not passing them as parameters allows vectorization.

```
s171(a,b,n)
float a[],b[];
 int n;
 { register int i;
  for (i=0; i<n; i++)
  a[i*n] = a[i*n] + b[i]; }

/* Call from another routine */
main()
{ float a[10000], b[10000];
  int n = 72;
  ...
  s171(a,b,n);
  ...
}
```

Figure 1: A function with arrays as parameters is NOT vectorized

The Argonne test suite attempts to test the effectiveness of compiler optimizations on local loop constructs, as opposed to interprocedural constructs and problems. Therefore, the parameter passing of arrays was eliminated by changing to global arrays in the C version with the Fortran version left unchanged.

```
float a[10000], b[10000];
 int n = 72;
s171()
{ register int i;
  for (i=0; i<n; i++)
  a[i*n] = a[i*n] + b[i]; }

/* Call from another routine */
main()
{ ...
  s171();
  ...
}
```

Figure 2: A function with global variable arrays IS vectorized

2. 1. Argonne test suite results

Initially, the global results for each architecture is described. The Argonne test suite breaks the loops into four sections: dependence analysis, vectorization techniques, idiom recognition and language completeness. Therefore, the results in each of these areas is analyzed in separate sections.

In the graphs that follow, The Y-axis is the number of kernels. The bars represent the number of kernels that fall into the particular category. Full vectorization means that all computation within the loop was fully vectorized. Partial vectorization means that the loop was split so that some of the computation was vectorized, but there was a scalar part of the loop that was not vectorized. Conditional vectorization means that two versions are present in the generated code, and at run-time the check is made to see whether the loop can be run in vector mode or not. Scalar means that the kernels remain in scalar mode and no vectorization occurred.

2. 1. 1. Cray results

Cray has two Fortran compiler products, one is **cft77** compiler, and the other is **cf77** which includes a preprocessor **fpp** in front of **cft77. fpp** is a dependence analyzer which assists in the vectorization and parallelization of Fortran codes. Figure 3 shows the comparison of the results of these two compiler packages. The dependence analyzer makes a large difference, allowing for substantially higher rates of vectorization. Therefore, the C compilers were compared to **cf77.** As one can see from Figure 4, the dependence analyzer made a difference in the results. In fact, the **cft77** results are strikingly similar to the Cray 2 **scc** results. One can conclude

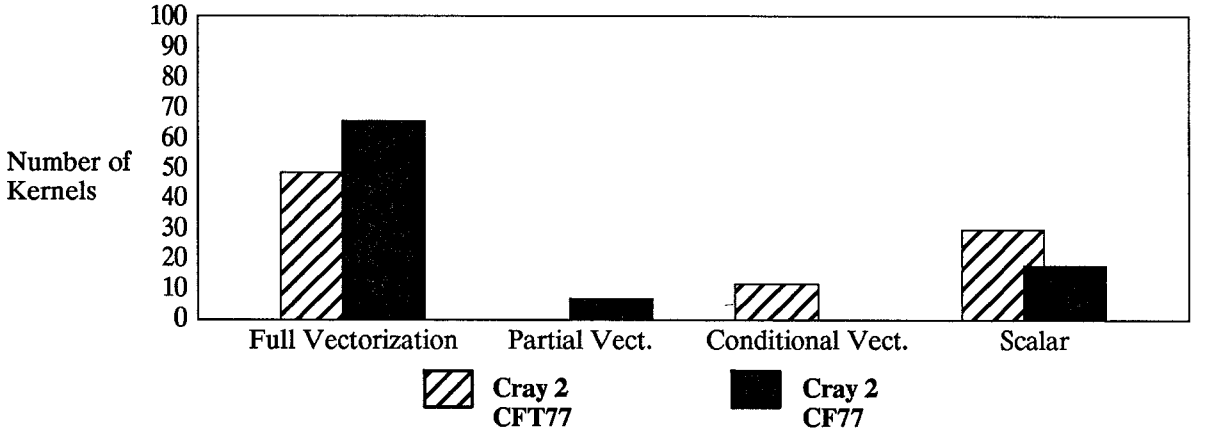

Figure 3: Cray 2 CF77-V4.0.3 vs. CFT77 V4.03 on the Argonne Test Suite

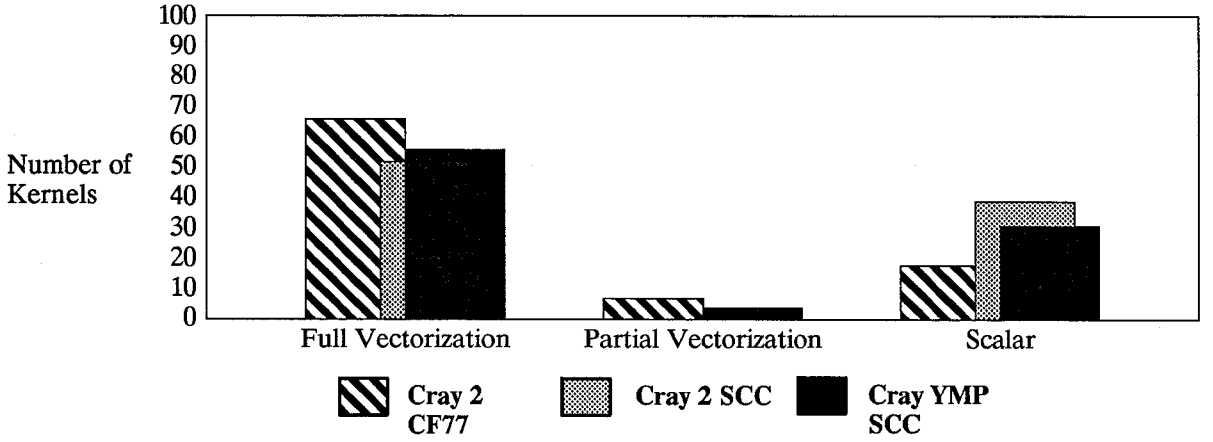

Figure 4: Cray 2 CF77-V4.0.3 vs. SCC 3.0 on the Argonne Test Suite

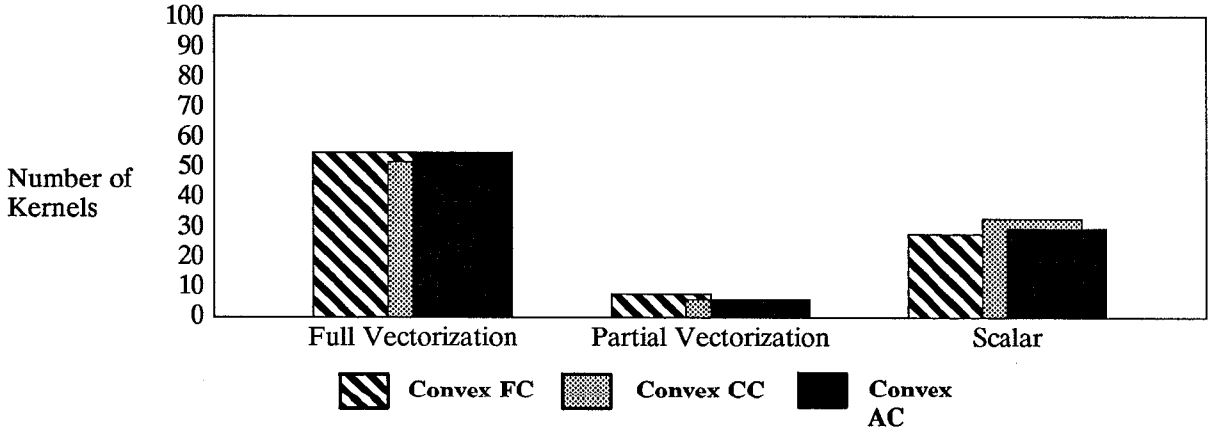

Figure 5: Convex FC 6.1 vs. Convex CC 4.1 and Convex C Application Compiler 1.0

that *a dependency analyzer can make significant improvements in vectorization and should be part of C vectorizing compilers.*

2. 1. 2. Convex results

Figure 5 shows the vectorization results of the suite. using the Convex compilers. The primary reason for the compilers to give similar results is the common backend. The Fortran version of the Convex Application Compiler was not used on this suite, but was used on the new version of the Fortran Argonne suite [Lev91], where some additional kernels that test interprocedural analysis and symbolic dependences were vectorized

2. 1. 3. Dependence analysis

The first part of the Argonne test suite looks at a broad area of loops that attempt to test the compiler's ability to do dependence analysis. This includes such things as linear dependence testing, induction variable recognition, flow analysis and symbolic dependences. The results are shown in Table 1.

	Full Vectorization	Partial Vectorization	No Vectorization
Convex fc	15	0	9
Convex cc	14	0	10
Convex ac	16	0	8
Cray cf77	20	1	3
Cray 2 scc	15	0	9
Cray Ymp scc	16	0	8

Table 1: Results of the dependence analysis section of the Argonne test suite

As one can see, the Convex compilers achieve the similar results, with the interprocedural analysis capability of the Application Compiler providing a slight edge. The weakest areas in the Convex compilers are induction variable recognition and dependence testing using FFT subscripting. The Convex compilers do not recognize induction variables when an induction variable is under both sides of an **if** statement.

The Cray C compilers do not do as well in comparison with either the Cray Fortran compiler or the Convex compilers. The areas where the Cray C compilers do not do well are linear dependence testing, and induction variable recognition. For example, transpose vectorization and lower triangular systems are not recognized with linear dependence testing. Interprocedural analysis is not done by any of the Cray compilers.

2. 1. 4. Vectorization techniques

The second part of the Argonne test suite examines a group of loops that test the compiler's ability to do some vectorization transformations. Some of the vectorization techniques tested are: statement reordering, loop distribution, loop interchange, node splitting, scalar and array expansion, conditional handling, and loop peeling. When some of these techniques are applied, partial vectorization can occur, where perhaps no vectorization is possible initially. Table 2 shows how well the Convex and Cray compilers can use these sophisticated techniques. As one can notice, more loops are partially vectorized, ranging from 55% to 74% vectorization on individual loops.

	Full Vectorization	Partial Vectorization	No Vectorization
Convex fc	24	6	5
Convex cc	24	6	5
Convex ac	25	5	5
Cray cf77	20	6	9
Cray 2 scc	15	0	20
Cray Ymp scc	18	4	13

Table 2: Results of the vectorization section of the Argonne test suite

The Convex compilers did very well with these techniques, far exceeding the results of the Cray compilers. The few cases that the Convex compilers do not handle are in the area of loop interchanging in the presence of **if** statements, and a case where loop peeling would have resulted in vectorization.

The Cray compilers did not do as well. The areas where both **cf77** and **scc** did poorly are: loop interchanging, scalar and array expansion, and loop peeling. The C compilers were also not able to handle simple statement reorderings, simple loop interchanges, and node splitting. Partial vectorization has only recently been added to the Cray C compilers, and the analysis has not reached the sophistication of the Convex analysis.

2. 1. 5. Idiom recognition

Programmers tend to use certain programming idioms frequently in their code. Therefore, compilers often need to recognize these programming idioms as special cases, optimizing and vectorizing in the presence of such idioms. The third part of the Argonne test suite tested the compilers' abilities in the area of idiom recognition. Idioms such as reductions, recurrences, min/max recognition and search loops are tested by this section. Table 3 gives the results from the Cray and Convex compilers. The Cray compilers did recognize most of the idioms, replacing the code with vectorized intrinsics.

	Full Vectorization	No Vectorization
Convex fc	8	7
Convex cc	7	8
Convex ac	7	8
Cray cf77	12	3
Cray 2 scc	11	4
Cray Ymp scc	11	4

Table 3: Results of the idiom recognition section of the Argonne test suite

This is one category where the Convex **fc** compiler does do a little better than the Convex C compilers. This is not surprising since the idioms tested are Fortran

idioms and would not normally be done the same way in C. The idioms that all the compilers could not handle are the search loops and recurrences; and the Convex compilers did not recognize min/max reductions.

2. 1. 6. Language completeness

For completeness, the Argonne test suite has as its last section a set of kernels that test language features and idioms that are primarily unique to Fortran. 17 of these kernels were left in to test the C compilers, primarily to see how they would handle some "non-normal C code". Results in Table 4 show that, indeed, the Fortran compilers are better.

The Convex compilers were surprisingly similar in their results. One critical area that the C compilers did not handle is intrinsics. Both the Cray and Convex Fortran compilers are able to recognize intrinsics and vectorize appropriately. C does not have intrinsics as part of the language, but there should be some mechanism to recognize such Fortran intrinsics (and other intrinsics), so that vectorization is not inhibited. Another situation that should be handled by all compilers, but is only handled by the Convex **fc** compiler is the presence of an I/O statement within a vectorizable loop. The **fc** compiler is able to partially vectorize in the presence of a simple print statement. None of the Convex compilers is able to handle **break** statements or **exit** calls within a vectorizable conditional loop.

	Full Vectorization	Partial Vectorization	No Vectorization
Convex fc	8	2	7
Convex cc	7	0	10
Convex ac	7	1	9
Cray cf77	14	0	3
Cray 2 scc	11	0	6
Cray Ymp scc	11	0	6

Table 4: Results of the language completion section of the Argonne test suite

The **cf77** compiler did very well in this section, only missing loops that have **exit** calls or **break** statements, and not partially vectorizing a loop that has a subroutine call in the middle of it. Ideally, interprocedural analysis would be present in the compiler, so this loop could be fully vectorized. However, it is possible to break apart the loop into sections and vectorize each section separately. The Cray C compilers fail in the same places as the Cray Fortran compiler, with the addition of one loop recognition kernel.

2. 2. Conclusions from the Argonne test suite

While the Argonne test suite was written to test the ability of Fortran compilers to vectorize, it has clearly pointed out strengths and weaknesses of vectorizing C compilers. This suite of kernels has demonstrated the strength of two language front-ends sharing optimization/vectorization techniques, like the Convex compilers. This commonality allowed both compilers to do as well as the other in many cases. This is not the case of the Cray compilers. The Cray compilation systems are moving to such a commonality, but currently, the **cf77** package has a powerful dependence analyzer that allows for much more vectorization. It is also interesting to note that there were also a few cases that the **scc** compilers do better than **cf77**, perhaps again a result of not sharing all optimization facets of the compilers.

More importantly, the Argonne test suite demonstrated areas that need to be improved and added to these vectorizing C compilers. Techniques that need to be added to one or all of the compilers are:

- Interprocedural Analysis
- Partial Vectorization around I/O
- Intrinsic Function Recognition
- Reduction Idioms
- Recurrences

The areas that need to be enhanced in C compilation systems are:

- Linear Dependences
- Induction Variable Recognition
- Loop Interchanges
- Node Splitting
- Scalar and Array Expansion
- Loop Peeling
- Statement Reordering

While this suite does test the ability of compilers to recognize and exploit opportunities for vectorization, it does not do any performance analysis to judge whether or not vectorization is the most efficient optimization for a given kernel. The new Fortran version 3.0 of the Argonne Suite [Lev91] attempts to address the issue for Fortran.

3. The C test suite

The Argonne test suite was written for Fortran compilers, and even with a C translation, it does not fully test C features. Therefore, an additional 40 kernels were developed to test certain C features. Many sophisticated pointer and structure features of C are not tested because vectorizing C compilers currently only handle the simplest of features. As C compilers get better, more kernels will be added to this suite.

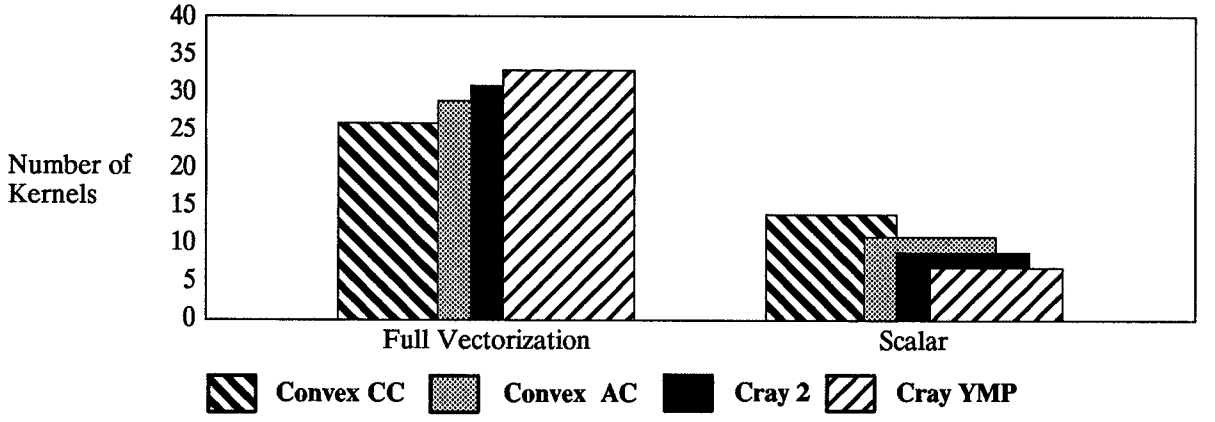

Figure 6: C Suite Results

3. 1. Results of the C suite

Figure 6 is a graph showing the overall results. Even though the numbers are similar, the Convex and Cray compilers do vectorize different types of loops from one another.

C has a larger variety of loop control structures than Fortran with **for** loops, **do while** loops, and **while** loops. Several variations of these loops were included in this suite. The Convex compilers do not vectorize any loop that has a **break** in it while the Cray compilers are able to vectorize all of the loop kernels.

One of the powerful features of C is the ability to create structure types and create structures of any form. Often these structures are dynamically allocated, allowing the programmer to develop dynamic data structures of any form. In this suite of kernels, the simplest of static structures are used to see if the C compilers recognize the static structures, their components, and are able to vectorize intelligently with such objects. Individual structure component accesses were vectorized, but implicit copying of two vectorizable structures is not done by either compiler. An example of this is given in Figure 7. It appears that the analysis in these compilers does not really understand structures. The simplest of structure copies does generate a block copy in the Convex compilers and a vectorized copy in the Cray compilers.

The most powerful feature of C is the pointer. Pointers are used for arrays, for strings, for structures, for dynamic allocation, for function parameters and results, and for just about everything else in C. So any compiler that is going to successfully vectorize C will have to handle pointers as best as possible. Since the ability to vectorize and parallelize in the presence of pointers [wor89] is currently an active research field, it is not surprising that current compilers are limited in their vectorization ability in the presence of pointers. Therefore, only a handful of the simplest pointer expressions are included as part of the C Suite.

```
struct manyn { int s;
               float ss;
               int v[100]
               float d[100]
             } ma[2000], mb[2000];

f202()
{ int i;
  for (i=0; i<2000; i++)
     ma[i] = mb[i];
}
```

Figure 7: Implicit many item copy

The Convex compilers did not vectorize any loops with C pointers used as loop bounds or induction variables. The Cray Y-MP compiler did vectorize in the presence of the simplest of these types of pointers while the Cray 2 compiler only recognized some. An example kernel that only the Cray Y-MP compiler vectorized is shown in Figure 8.

```
int a[2000], b[2000];
f303()
{ int *p, i, n;

  n=0;
  p=&n;
  while (*p < 2000) {
     a[*p] = b[*p];
     (*p)++;
  } }
```

Figure 8: A pointer as a condition on a loop

The Convex Application compiler recognizes simple pointer notation for some array accesses, while the Convex **cc** compiler was considerably weaker in this area. However, only the Cray compilers were able to recognize the pointer notation when **&** had been used to set up the pointers. This is shown in Figure 9.

```
float x[2000, y[2000], z[2000];
f503(alpha)
float alpha;
{ register int  i, n=ILEN;
 float *p, *q, *r;
 p = &x[0];
 q = &y[0];
 r = &z[0];
 for (;n;n--)
  *p++ = *q++ + alpha * *r++;
}
```

Figure 9: Using pointer array notation

The Convex compilers recognizes and vectorizes more kernels with character arrays than the Cray compilers. One example is shown in Figure 10.

```
char c1[256], c2[256], c3[256];
int  i;
for (i=0; i<256; i++) {
  c1[i] = i;
  c2[i] = 255 - i;
  c3[i] = 'z' - (i % 10);
 }
```

Figure 10: Character array initialization

The Convex application compiler was able to vectorize a dynamically allocated character array that was considered too complicated by the other compilers. This is shown in Figure 11.

```
char *c5, c1[256], c2[256];
 int i;
c5 = (char *)(calloc(256,sizeof(char *)));
for (i=0; i<256; i++)
    c5[i] = (c1[i] & c2[i]) << 2;
```

Figure 11: Use of dynamically allocated char arrays

The interprocedural analysis capabilities enable the Convex application compiler to vectorize a kernel which includes a function call in the loop. Interestingly enough, the Cray compilers recognize that the value returned form the function call is superfluous, but still does not vectorize around it. This example is shown in Figure 12.

```
unsigned int ai[ILEN], bi[ILEN];
int i;
{ ...
for (i=0; i<ILEN; i++)
    ai[i] = (foo(i*i),(bi[i] << 2)) + 1;
}
foo(m)
int m;
{ return(m+1) }
```

Figure 12: Function call with , operator

Several recurrences were not broken by any of the compilers, including one described by [AJ88]. The Cray 2 compiler did not vectorize in the presence of a simple doubling recurrence that the Cray YMP compiler recognized.

3. 2. Conclusions from the C suite

The C suite was written to try to enhance the Argonne Suite. Currently, only the simplest of C structures are tested, and the suite needs to be enhanced with more sophisticated control structures, pointer uses and data structures. However, the suite does show some additional areas in the vectorizing C compilers that need to be improved upon. Both compilers need to add the following features to their vectorization analysis:

- Pointer Analysis
- Recognition of Pointers as Induction Variables
- Static Structure Analysis
- Recurrence detection and breaking

Pointer and structure analysis are current areas of research [wor89], with some of the main problems being alias detection between pointers, and pointer arithmetic.

4. A suite of C kernels from applications

The above two suites of C kernels were created by compiler researchers interested in testing vectorizing compilers. So several questions are: How do such results apply to actual C applications? How do application programmers write C programs for vectorizing computer architectures? How well do the compilers vectorize C code written by application programmers?

In response to these questions, a suite of 44 kernels was developed from examining real C applications. All of the applications currently run on either a Convex or a Cray machine. As more and more applications are examined, this suite continues to grow.

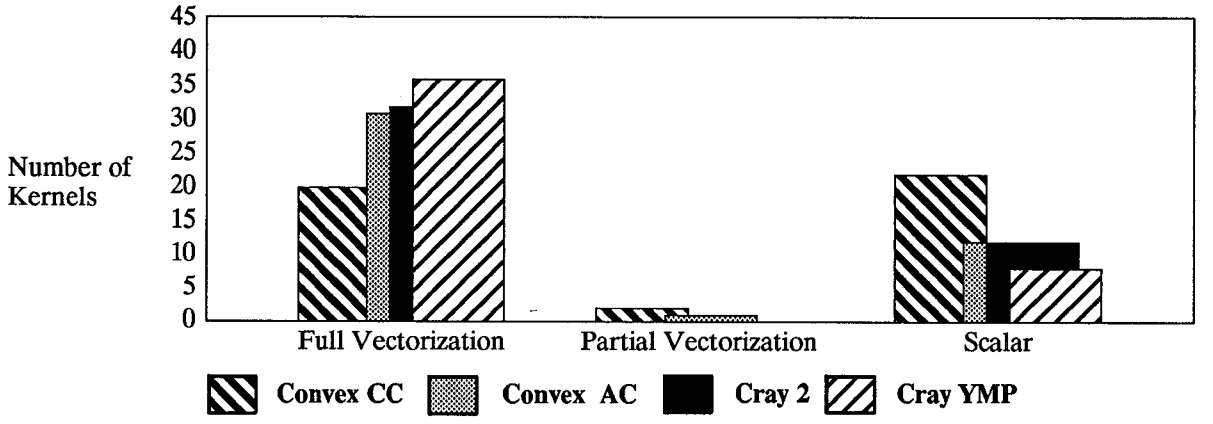

Figure 13: Application Suite Results

4. 1. Results from C Application Suite

The results from this suite appear in Figure 13. The high number of vectorized kernels is encouraging since these are from real applications.

The Convex Application Compiler successfully recognizes some simple structure references that the Convex **cc** compiler does not understand. The main reason for the dramatic difference between the Convex compilers is the inability of **cc** to vectorize in the presence of dynamically allocated arrays (The use of directives helps solve this problem). Dynamic allocation is heavily used in many applications because it is a means of limiting the amount of memory used by depending on actual run-time data. Most codes examined dynamically allocated the arrays and then used array notation. A sample of this is shown in Figure 14.

```
int *f, *g, si, i;
si = 3000;
f = (int *)malloc(si*sizeof(int));
g = (int *)(malloc(si*sizeof(int));
...
for (i=0; i<3000; i++)
  f[i]=g[i];
}
```

Figure 14: Dynamic allocation of arrays

The Cray 2 compiler does not vectorize in the presence of a macro as a loop bound, while this works on the Cray Y-MP. The Cray 2 compiler also does not recognize static structure accesses as well, inhibiting vectorization.

The Cray compilers are able to vectorize a loop with a simple **switch** statement in it while the Convex compilers can not.

The appearance of the **address of** operator (&) in a loop also inhibited the Convex compilers from vectorizing.

The Convex compilers also did not vectorize when an **unsigned** index was used in a loop as the induction variable.

Neither Convex compiler was able to handle pointer notation to arrays of longs, while the Cray compilers were able to vectorize these loops. An example is given in Figure 15.

```
n151(a,b,c,n)
long a[], b[], c[], n;
{ long *lim;
  n=MMAX;
 lim=a+n;
while (a<lim)
    *c++ =*a++ &~ *b++;
}
```

Figure 15: Pointer notation

The Convex compilers are able to do more partial vectorization than the Cray compilers, doing more sophisticated loop splitting. The Convex compilers are also able to vectorize a character loop that the Cray compilers do not handle (**scc 3.0** is the first version to vectorize character arrays, so the analysis is not as complete).

None of the compilers were able to vectorize loops that used pointer notation to access character arrays. This is shown in Figure 16.

```
char sent[1000];
char *pc;
...
for (pc=sent; *pc; pc++)
  *pc = *pc +2;
```

Figure 16: Pointer notation with char arrays

Recurrences with logical operators were not broken by any of the compilers. Nor were general search loops of character arrays vectorized by either vendor.

Some simple structure accesses are recognized by both compilers, but as the C suite demonstrated, more complicated accesses inhibit vectorization. In fact, when data structures are initialized in loops containing other simple array initializations, the whole loop is inhibited from vectorization when indeed partial vectorization could be done.

4. 2. Conclusions from the application suite

From developing the Application Suite, some interesting conclusions were reached on both the ability of the compilers to vectorize certain kernels as well as some insight into some C application programs. This suite is far from comprehensive, and is being added to as frequently as is possible. The key areas that need to be improved and added to vectorizing C compilers are:

- Analysis of dynamically allocated arrays
- Pointers to Structures and Characters

In examining the applications to extract out kernels and interesting vectorizable pieces of code, one quickly discovered that many application programmers have attempted to work around the deficiencies of the vectorizing C compilers. Temporaries are added in to make loop control simpler, scalars are hand expanded, loops are split apart by the programmer, and various other tricks are used. While this lets the programmer get much better efficiency, it does lead to perhaps more obscure code. One also noticed that application programmers tend to use more array notation and more of a "Fortran" style in their C codes than general C codes. This should make the job of vectorizing compilers easier for application codes.

5. Conclusions and future work

Testing the vectorizing C compilers with these suites of programs has proven very useful. The compilers actually proved to be better at vectorization than expected. Many vectorizable loops were vectorized, some pointer notation is recognized, and other C constructs are handled. Much of the vectorizing Fortran technology has easily been applied to C compilers.

The advice to the programmer is to understand the particular architecture and compiler being used. Many of the loops presented are often in programmer's code, and if the programmer understands what can and cannot be done by the compiler, better code can be produced. This paper has demonstrated the strengths and weaknesses of the Cray C and Convex C compilers, and should be able to be used as a guide. The programmer also needs to study the compiler options and directives carefully, since as this study shows, the compilers are not finding all vectorization opportunities.

The Convex C Application Compiler appears to be significantly more sophisticated in analysis than the **cc** compiler. The interprocedural analysis in this compiler was not well tested by these suites, but it did succeed in a few places. The pointer and symbolic analysis were tested by these suites and the results were encouraging.

The Cray compilers have improved since the initial study [Smi90]. Partial vectorization and character array vectorization is a step in the right direction. However, interprocedural analysis needs to be added. Currently, there is an "inline" option that allows some additional optimization.

Each section provided a list of features that needed to be improved and/or added to the current C vectorizing compilers. These lists will hopefully assist vendors in improving their compilers. The source of these C kernels is available electronically. Please contact the author at llsmith@super.org.

Future work will be done on improving the suites, and tracking the results as newer versions of the compilers are produced. Attempts will also be made to examine the vectorizing C compilers of other vendors. However, one question when looking at kernels, is how well do the kernels represent an application load? Therefore, future work will be in developing empirical studies of the usage of C in scientific applications. From these studies, it is hoped that compiler writers will obtain a better idea of what analysis techniques might prove most effective.

One weakness of these suites of kernels is that the quality of code generated is not judged and the performance of the resulting code is not measured. In some cases, vectorization might be the least optimal way of generating code. Therefore, future work also needs to include some measure of code quality.

This work was done to understand the abilities of optimizing/vectorizing C compilers. However, it was also done to see what will be needed for parallelizing C compilers. Vectorization can be viewed as a simple form of parallelization (fine-grained), so all of the same problems with vectorization occur with parallelization. However, the problem becomes magnified, so that issues such as interprocedural analysis, alias analysis and pointer analysis become more critical. Vectorization is that first step, and it is encouraging that at least Cray and Convex have the foundations of good vectorizing C compilers.

Bibliography

[AJ88] Randy Allen and Steve Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 241–249, Atlanta, Georgia, June 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers 1988.

[CDL88] David Callahan, Jack Dongarra, and David Levine. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the Supercomputing '88*, pages 98–105, Orlando, Florida, November 1988. IEEE Computer Society Press..

[Con91a] Convex Computer Corporation, Richardson, TX. *CONVEX C Guide*, 1991.

[Con91b] Convex Computer Corporation, Richardson, TX. *CONVEX C Optimization Guide*, 1991.

[Con91c] Convex Computer Corporation, Richardson, TX. *CONVEX Application Compiler User's Guide*, 1991.

[Cra90] Cray Research Inc., Mendota Heights, MN. *Cray Standard C Programmer's Reference Manual.* SR-2074 2.0, 1990.

[Lev91] David Levine. Test Suite for Vectorizing Compilers. Version 3.0, January 1991. .

[Nob89] Hiromu Nobayashi. A Comparison Study of Automatically Vectorizing Fortran Compilers. *Vector Register*, pages 3–8, March 1989. Translated by: Christopher Eoyang.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[Smi90] Lauren L. Smith. An Analysis of the Vectorization Abilities of Current C Compilers. July 1990. Releasable Version of an internal Dept. of Defense technical report, TR–R53–05–90..

[Wol82] Michael J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, University of Illinois, October 1982.

[wor89] *Workshop on Parallelism in the Presence of Pointers and Dynamically–Allocated Objects*, Bowie, Maryland, March 1990. Supercomputing Research Center. Technical Note SRC–TN–90–292.