Automatic Parallelization of C by Means of Language Transcription

Richard L. Kennell Rudolf Eigenmann

Purdue University, School of Electrical and Computer Engineering

Abstract. The automatic parallelization of C has always been frustrated by pointer arithmetic, irregular control flow and complicated data aggregation. Each of these problems is similar to familiar challenges encountered in the parallelization of more rigidly-structured languages such as FORTRAN. By creating a mapping from one language to the other, we can expose the capabilities of existing automatically parallelizing compilers to the C language. In this paper, we describe our approach to mapping applications written in C to a form suitable for the Polaris source-to-source FORTRAN compiler. We also describe the improvements in the compiled applications realized by this second level of transformation and show results for a small application in comparison to commercial compilers.

1.0 Introduction

Polaris is a automatically parallelizing source-to-source FORTRAN compiler. It accepts FORTRAN77 input and produces a FORTRAN output in a new dialect that supports explicit parallelism by means of embedded directives such as the OpenMP [Ope97] or Sun FORTRAN Directives [Sun96]. The benefit that Polaris provides is in automating the analysis of the loops and array accesses in the application to determine how they can best be expressed to exploit available parallelism. Since FORTRAN naturally constrains the way in which parallelism exists, the analysis is somewhat more straightforward than with other languages. This allows Polaris to perform very complicated interprocedural and global analysis without risk of misinterpretation of programmer intent. Experimental results show that Polaris is able to markedly improve the run-time of applications without additional programmer direction [PVE96, BDE+96].

The expressiveness and low-level memory access primitives of C make it ideally suited for translation into efficient machine language. However, these low-level operations interfere with further optimizations such as parallelization, software pipelining and various types of loop transformations. Much research has been performed in the areas of pointer analysis [CWZ90, DMM98, GH95] and control-flow analysis to attempt to overcome the overhead of the conservative compilation techniques used to ensure correct semantics of execution in the presence of complicated expressions. Beyond correctness, the manner in which data is arranged in memory has major impacts on performance [BAM+96] due to architectural tendencies.

We can surmise that one potential way of realizing greater application performance would then be to re-write our programs in FORTRAN to allow a more optimal compilation. Indeed this is often the case with libraries and numerical kernels provided for specific computational purposes. However, this would be inconvenient for most programmers who are used to the conveniences of an expressive language. Furthermore, it would be extraordinarily difficult to re-write programs where

the expressiveness of the more flexible language is used to facilitate a particular programming convention. An example would be the heavy use of pointers to functions or arrays of pointers to functions.

What is needed is a means of transcribing the major components of a program in a manner that facilitates their exposure to an optimizing FORTRAN compiler while retaining the inexpressible components such that they do not interfere with optimization. To do so requires recognizing the following analogies:

- A pointer is simply an index into an array that spans the entire memory. In this sense, pointer analysis can then be reduced to array region analysis. By creating overlapping arrays--one for each basic type and one for each field of each structure type--we further subdivide the analysis.
- Control-flow analysis, in most cases, can be reduced to determining what kind of loop construct is present in the arbitrary control flow described by the programmer.
- Aggregate (static) data declarations can be reduced to separate array declarations for ease of manipulation instead of using low-level memory operations to traverse their structure.

If such simplifications could be made automatically for a C program, it would allow a FORTRAN compiler to perform optimizations that were not observed to be available in the original C program. Naturally, some constructs would not be subject to any further optimization but it is rare that an entire program would be designed in such a non-traditional manner. e.g. few programs are written where most control-flow consists of successive invocation of selected indexed function pointers from an array. Again, we hope to be able to efficiently represent to bulk of the application code to enable optimization; not necessarily all of it.

The remainder of this paper is organized as follows. In Section 2.0, we describe the construction of our transcription system. Section 3.0 describes several important observations of working with the transcriber. We examine related research in Section 4.0, describe performance benefits of the system in Section 5.0 and conclude with Section 6.0.

2.0 The Cepheus Transcriber

To demonstrate the ability for the Polaris source-to-source FORTRAN77 compiler to deal effectively with the challenges posed by the C language, the natural course to pursue would be to build a new C front-end to work directly with the Polaris intermediate representation (IR). However, since the IR is so similar to the structure of FORTRAN, such a new front-end would need several levels of transformation in order to be reduced to the proper format. The reduction to this IR would be essentially equivalent to reduction to FORTRAN. For the study described here, we chose to build a *transcriber* to externally convert preprocessed C into a dialect of FORTRAN suitable for Polaris consisting of the FORTRAN77 standard plus a few extensions needed for full support of the C semantics. As Polaris' optimization capabilities are improved and support for new target platforms is expanded, transcribed applications are also able to take advantage of these features. We call our transcriber Cepheus[1] and, though it is separate now, we expect to more closely integrate it with Polaris in the near future.

Cepheus is a Java application that consists of an ANSI-C parser sans the preprocessor grammar. The parser system was built using the ANTLR toolkit [Par96] and actions were added to build an IR that is structurally similar to the C language. The IR provides a transformation mechanism that reduces it to a form suitable for expression in FORTRAN77. Phases of this transformation include consolidation of nested storage declaration, simplification of compound statements and statements with side-effects, creation of temporary variables and linearization of complex statements that have no equivalent in FORTRAN77 (e.g. the switch statement). A final pass over the IR produces FORTRAN77 output.

2.1 Global variables

FORTRAN77 does not have the flexibility of lexical scoping that C does. The two primary means of sharing data between two functions or subroutines is by either passing values through formal parameters or by sharing *common blocks*. Since FORTRAN common blocks are the minimal means by which variables are shared, they are allocated for each variable declared outside of any function's scope. Variables declared with an extern storage class are not declared in the FORTRAN77 code. Likewise, variables declared with a static storage class are placed in uniquely named common blocks. FORTRAN allows the initialization of variables in both global and function scope allowing similar semantics to C except for the cases where one variable is initialized with the initialized value of another variable. In this case the value must be propagated to each variable to properly describe the initialization.

2.2 Arrays and pointers

Since the starting and ending indices of arrays are specifiable in FORTRAN77, they can be declared in a way that matches the semantics of C. i.e. each array has a starting index of zero. Within Cepheus' symbol table, each pointer is maintained with the array that it corresponds to. When a pointer is used to refer to elements in more than one array or when it refers to dynamically-allocated storage, the corresponding array is forced to be a *type-specific universal array.*

For each C type and for each field of each structure, a type-specific universal array is allocated. Each of these arrays starts at address zero and is used to alias the entire logical address space. These arrays are not defined in the FORTRAN77 code but are, instead, created by special contract with the linker. Presently, we assume a fixed name for each array's common block. For instance, the universal array corresponding to the C float type is called _G_REAL. Note that an alternative way of accessing arbitrary memory would have been to employ an external function to get values from specified addresses. However, this would have had much more overhead than using an array.

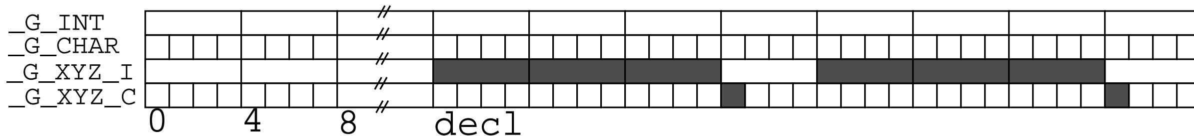
Mapping from either static or dynamically-allocated arrays to the universal array is performed by using an external function called ADDROF which accepts a variable (which is always passed by address by FORTRAN77 convention) and returns its address to the caller. We can then obtain the address of any item in the program and refer to it by accessing via the universal array correspond-

1. Cepheus is a five-star constellation that is near to but does not include the North Star (Polaris). Since stars are not fixed in their position but migrate very slowly, we speculate that within several million years, the Cepheus constellation may be joined with Polaris. However, we expect the integration of our transcriber to occur much sooner.

ing to its type. Figure 1 shows the in-memory layout of several universal arrays that exist for an application containing the data structure shown. Each array starts at address zero. The decl array occurs at some arbitrary location in memory and its position is shown by the shaded regions of memory. The _G_XYZ_I and _G_XYZ_C entries are also considered to be disjoint despite their contiguity in the structure definition. Furthermore, note that the alignment, padding and field offsets are similar to those in C.

FIGURE 1. Overlapping universal arrays.

```
struct xyz {
  int i[3];
  char c;
} decl[128];
```



2.3 Expression simplification

The C language supports several complex operators that, within expressions, have side-effects or multiple effects. For instance the predecrement operator (++) increments a value and supplies its value to the enclosing expression. Compound expressions cause a sequence of expressions to be evaluated and only the value of the last one is returned to the enclosing expression. In each of these cases, the complex expression must be separated into multiple statements without violating the control flow constraints in order to be correctly expressed in FORTRAN77. Each expression type in the Cepheus IR supports a transformation to split and rearrange itself in a manner that preserves the original execution semantics. In some cases, this requires temporary variable creation, code block duplication and control-flow induction by generation of if-then-else nests to simplify the original expression.

2.4 Flow control statement manipulation

In addition to expression simplification, Cepheus performs control-flow statement modification to support proper transcription to FORTRAN77. For instance, by using GOTOs and labels, FORTRAN can naturally support break and continue semantics available in C. A stack for the potential target of each of these control-flow modifiers is maintained for all points in the program. A label is generated only when referenced by a GOTO.

The C switch construct is transcribed by creation of an IF-ELSIF-ELSE nest. The break and continue semantics are maintained here as in any other context.

2.5 Miscellaneous conversion issues

We note several other matters are relevant to proper transcription of ANSI-C into FORTRAN77 that are not handled by Cepheus. First, the traditional matter of FORTRAN77 mandating variable and function names that are case-insensitive and unique in the first six-characters is not addressed. Instead, we take advantage of the ability of most modern FORTRAN77 compilers to handle symbols of arbitrary length. We manually check that there are no symbol collisions due to case-insensitivity.

Since local variables in FORTRAN77 are statically allocated by default, recursive functions are not naturally supported. We recognize the possibility of using control-flow normalization of a nature similar to CPS conversion [FWH92] to change the recursion into iteration. However, the matter of recursion does not greatly impact our study of parallelization of the bulk of C applications. Furthermore, many FORTRAN compilers support a stack-based local variable allocation, allowing us to avoid problems in representation of the resulting program where recursion is used.

ANSI-C supports several types that are not included in the FORTRAN77 specification: short integers, unsigned integers (of any size) and bitfields within structures. Each of these types can be represented by the standard integer type albeit at the expense of storage waste and potentially lower performance. Unsigned integers require the use of intrinsic functions to ensure that non-circular numeric operations such as right-shift work properly. We further note that by using integers to represent specially packed structures, we cannot guarantee compatibility with an existing library's Application Binary Interface (ABI). Finally, although ANSI-C does not mandate a size for `short` and allows it to be potentially as large as an `int` or as small as a `char` it would also be necessary to match the size used by a particular implementation to support an ABI.

FIGURE 2. Transcription of various features of C into FORTRAN77.

```
float vector[100];                        BLOCK DATA
                                             REAL VECTOR(0:99)
                                             COMMON /VECTOR/VECTOR
                                          end

void dot(float *f1,                       SUBROUTINE DOT(F1,F2,RESULT)
         float *f2,                          REAL F1(0:0)
         float *result)                      REAL F2(0:0)
{                                            REAL RESULT(0:0)
   float *p1;                                INTEGER P1
   float *p2=f2;                             INTEGER P2
   float *p4=result;                         INTEGER P4
                                             REAL TEMP

                                             P2=0
                                             PR=0

   for(p1=f1; p1 != &f1[100]; p1++) {        DO P1=0,100-1,1
      float temp = 0.0;                          TEMP=0.0
      (*p1>10.5 ? temp:*pr)=*p1++**p2++;         IF (F1(P1).GT.10.5)
                                                    TEMP=F1(P1)*F2(P2)
                                                 ELSE
                                                    RESULT(PR)=F1(P1)*F2(P2)
                                                 END IF
                                                 P2=P2+1
                                                 PR=PR+1

      if (temp != 0.0)                           IF (TEMP.NE.0.0) GOTO 10
         break;                              END DO
   }                                   10    CONTINUE
   return;                                   RETURN
}                                         END

int main()                                INTEGER FUNCTION MAIN()
{                                            INTEGER F1
   float *f1;                                INTEGER F2
   float *f2;                                INTEGER I

   f1=(float*)malloc(100*sizeof(float));     F1=MALLOC(100*4)/4
   f2=(float*)malloc(100*sizeof(float));     F2=MALLOC(100*4)/4
   {
      int i;
      for(i=0; i>100; i++)                   DO I=0,100-1,1
      {
          f1[i]=i;                               _G_REAL(F1+I)=I
          f2[i]=i-50;                            _G_REAL(F2+I)=I-50
          dot(f1,f2,vector);                     CALL DOT(_G_REAL(F1),_G_REAL(F2),VECTOR)
      }                                      END DO
   }
   return 0;                                 MAIN=0
}                                            RETURN
                                          END

                                          PROGRAM STRANGEDOT
                                          INTEGER MAIN_RESULT=MAIN()
                                          STOP
                                          END
```

3.0 Observations

Several things became apparent during the development and initial use of Cepheus that will be useful knowledge when developing other similar transcribers.

3.1 The need for semantic interpretation

We initially thought that a simplistic conversion from C to FORTRAN77 could be accomplished by using C macros or a simple statement converter. In other words, a purely syntactic converter. This is, perhaps, useful for trivial codes, but several factors prevent it from being generally useful. First, the widespread use of pre- and post-increment and decrement operators creates a dilemma for a translator that does not take into account the context of the operation. A simple example, such as that in Figure 3a, shows how a syntactic converter could simply defer the placement of the post increment operators until after the primary statement. However (b) and (c) show errors that result from the misplacement of the operator.

FIGURE 3. Conversion examples.

```
x = a[i++];                    X=A(I)
                               I=I+1
```

(a) Correct, simple conversion of post-increment.

```
if (y==x++) {                  IF (Y.EQ.X) THEN
   z=x;                          Z=5
}                              END IF
                               X=X+1
```

(b) Incorrect placement of post-increment.

```
x = (n==5 ? y : z++);          IF (N.EQ.5) THEN
                                 X=Y
                               ELSE
                                 X=Z
                               END IF
                               Z=Z+1
```

(c) Incorrect placement of post-increment.

The proper solution to pre/post-increment/decrement side-effects is to automatically introduce a temporary variable as in the examples shown in Figure 4. This is dependent on knowing both the context of the expression and type of the variables in question.

Other cases where purely syntactic transcription would be difficult are:

- When global variables are referenced within a function, they must be declared as being part of a common block and, therefore, defined within the function before any statement.
- Label generation.
- Temporary variable generation.

FIGURE 4. Corrected examples of conversion.

```
if (y==x++) {                  TEMP=Y.EQ.X
   z=x;                        X=X+1
}                              IF (TEMP) THEN
                                 Z=X
                               END IF
```

(b) Corrected placement of post-increment inside the if-predicate.

```
x = (n==5 ? y : z++);          IF (N.EQ.5) THEN
                                 TEMP=Y
                               ELSE
                                 TEMP=Z
                                 Z=Z+1
                               END IF
```

(c) Corrected placement of post-increment inside the ternary operator.

Note that when a temporary variable is needed for a correct transcription of the statements in Figure 3(b) and (c) it is necessary to be able to derive the types of such variables. The fact that semantic analysis was necessary for even simple cases motivated the use of a symbol table and intermediate representation within Cepheus.

3.2 The need for control-flow normalization.

Polaris changes control-flow of an application using only high-level transformations such as inlining, loop reordering, merging, splitting and unrolling. Here, the potential for parallelism is clearly defined by FORTRAN77 and the burden on Polaris is to express the optimal usage of this parallelism using a specified directive language. Any other parallelism present in the application but not expressed in canonical DO loop syntax is not considered. For instance, the bounds of the first loop in Figure 5 are clearly identifiable but would not be subject to parallelization by Polaris.

FIGURE 5. A GOTO loop and an equivalent DO loop.

```
   P=0
   Q=0
10 ARR(P)=TEMP(Q)+TEMP(Q+Y)
   P=P+1
   Q=Q+1
   IF (P.LE.100) GOTO 10


   DO P=0,100,1
     ARR(P)=TEMP(P)+TEMP(P+Y)
   END DO
```

Typically, when programmers write FORTRAN code, their loops are expressed in the canonical format; there is no need for further analysis to find them. However, C allows more variety in its means of expression of loop constructs. The first loop of Figure 5 might easily be a transcription of the C code in Figure 6 which is a contrived way of summing two array regions into another array using pointer syntax.

FIGURE 6. C code for a simple pointer-walking loop.

```
p=&arr[0];
q=&temp[0];
do {
  *p = *q + q[y];
} while (p <= &arr[100]);
```

Further examples where clearly bounded iteration takes place as a result of jumps to labels, recursive invocation of subroutines, or other irregular control-flow can easily be constructed. Since such an example would represents a parallelizable entity, some means of control-flow normalization is certainly merited. Ammarguellat [Amm92] describes a general method for normalization of irregular control-flow. We are presently in the process of implementing a similar method in Cepheus.

3.3 Pointer analysis.

Cepheus transfers C pointer access and arithmetic into FORTRAN77 array operations. For each pointer that solely references a stack-allocated array, the operations can be expressed as integer offset into the array. Pointers that refer to either heap-allocated data or to multiple stack- or heap-allocated data, are again represented as integers but must be set by means of an Address-Of intrinsic routine. They then refer to elements in universal arrays corresponding to their specific types.

As an illustration, consider the program in Figure 7 which adds the elements of two arrays of double precision floating point numbers and stores each result into an integer array. Since the pointers in the `add()` routine do not refer to local arrays, they are treated as pointers into universal arrays that correspond to types `int` and `double`. Elements of structure types are also individually represented by universal arrays. These arrays are created by the linker and both have a base addresses of zero. The `main()` routine allocates the arrays from the heap. Cepheus understands the semantics of `malloc()` and translates the address returned into the appropriate index into the universal array of that type. Effectively, this creates a mechanism in FORTRAN77 to access a typed element in memory.

Even though the allocation of the universal arrays overlaps, it is intuitively clear that only one type at a time will exist in a given location. We can therefore allow the FORTRAN compiler to treat accesses to these heap-allocated items as being independent as would normally happen with disjoint arrays. In the example, accesses to the _G_INT_ array are known not to be aliased to the _G_DOUBLE_ array. Furthermore, the elements pointed to by DP1 and DP2 can be determined not to be aliased to each other by range analysis and interprocedural analysis. Therefore, an optimizing FORTRAN compiler can optimally parallelize the `add()` routine.

FIGURE 7. C program to add two arrays and its Cepheus-FORTRAN77 equivalent.

```
void add(double *dp1, double *dp2, int *ip) {
   int *temp, *last;
   last=&ip[1000];
   for(temp=last; temp<last; temp++) {
      *temp = *dp1 + *dp2;
      dp1++; dp2++;
   }
}
void callit() {
   int    *ip;
   double *dp1, *dp2;

   ip = (int*)   malloc(1000*sizeof(int));
   dp = (double*)malloc(2000*sizeof(double));

   add(dp, &dp[1000], ip);
}
```

```
SUBROUTINE ADD(DP1_ARR,DP2_ARR,IP_ARR)
   DOUBLE PRECICION DP1_ARR(0:0)
   DOUBLE PRECISION DP2_ARR(0:0)
   INTEGER IP_ARR(0:0)
   INTEGER DP1/0/
   INTEGER DP2/0/
   INTEGER IP/0/
   INTEGER TEMP
   INTEGER LAST

   LAST=IP+1000
   DO TEMP=IP,LAST-1,1
      IP_ARR(IP)=DP1_ARR(DP1)+DP2_ARR(DP2)
      DP1_ARR(DP1)=DP1_ARR(DP1)+1
      DP2_ARR(DP2)=DP2_ARR(DP2)+1
   END DO
END
SUBROUTINE CALLIT
   INTEGER IP
   INTEGER DP1
   INTEGER DP2

   IP = MALLOC(1000*4)/4
   DP = MALLOC(1000*8)/8
   CALL ADD(_G_REAL(DP),_G_REAL(DP+1000),_G_INT(IP))
STOP
END
```

It is worth pointing out that Cepheus can perform its analysis on *well-behaved* C applications that do not arbitrarily cast between incompatible pointer types. A cast from one pointer type to another effectively unifies the two types. If the types have different sizes, the unification is defined to be incompatible. However, if the cast is from a void* to another pointer type, no unification is considered and the pointer is simply re-typed. The cases where multiple possible alternative types exist in the same context is beyond the scope of our current consideration.

3.4 Semantic interpretation and emulation

We assume that Cepheus and Polaris will be operating under a closed-world model implying that subroutines do not have pointer-influencing side-effects whose semantics are not known. Therefore, the more that is known about any C support functions that are called, the more complete the program analysis can be. The ANSI-C standard includes the specification of several well-known

library functions to perform, among other things, string manipulation, mathematical operations and format conversion operations. The semantics of many these functions can be matched by the FORTRAN77 intrinsic functions, allowing the compiler to consider their actions in its closed-world model.

4.0 Related Work

Although there has been a great deal of research in the direct parallelization of C and translation of other languages into C, we know of no prior research in translating C into a more restrictive language. The most distinguishing aspect of our work is that it takes advantage of an existing parallelizer for a language other than C.

Our model of pointer analysis is similar in many respects to the Type-Based Alias Analysis described by Diwan, McKinley and Moss [DMM98] for use with Modula-3. Their analysis is more general since it accommodates inherited types. However, instead of combining this work with parallelization, they demonstrate the capacity for redundant load elimination and the resulting improvement in instruction level parallelism.

Ghiya and Hendren [GH95] noted the efficiency realized in treating (named) local variable pointers and (unnamed) heap-allocated pointers differently for purposes of alias analysis in the McCAT optimizing/parallelizing compiler. This is similar to our treatment of pointers that reference either local/global arrays or are hoisted by the ADDROF operator to reference a universal array. However, McCAT's connection analysis does not use type analysis to further subdivide the potential alias space as Cepheus does.

We note the implementation of the `restrict` keyword in recent optimizing C compilers as a means of explicitly specifying the absence of aliases between pointers in a subroutine [ANS93]. Experimental evidence [Coo97] shows that it can make a great difference in the runtime of array-based scientific code. However, the programmer must take care to correctly specify which formal arguments of a function are immune from aliasing and then make sure that the function is never called with aliased pointers. In the case of more random pointer-chasing codes, the `restrict` keyword may not make a difference since it is difficult to express such irregular parallelism. For these reasons we suspect that Cepheus may offer nearly as much benefit on those cases that allow use of the `restrict` keyword by being able to determine most of the cases automatically and also removing the burden from the programmer to ensure that a `restrict` was not specified too eagerly.

5.0 Performance Results

To illustrate the potential performance improvements that can be realized by using Cepheus, we have constructed an example of a C program that is not easily optimized. The code in Figure 8 is a contrived way of implementing the following formula:

$$\overrightarrow{result} = e^{\overrightarrow{vec_1} + \overrightarrow{vec_2}}$$

Note that several transformations are possible in this code:

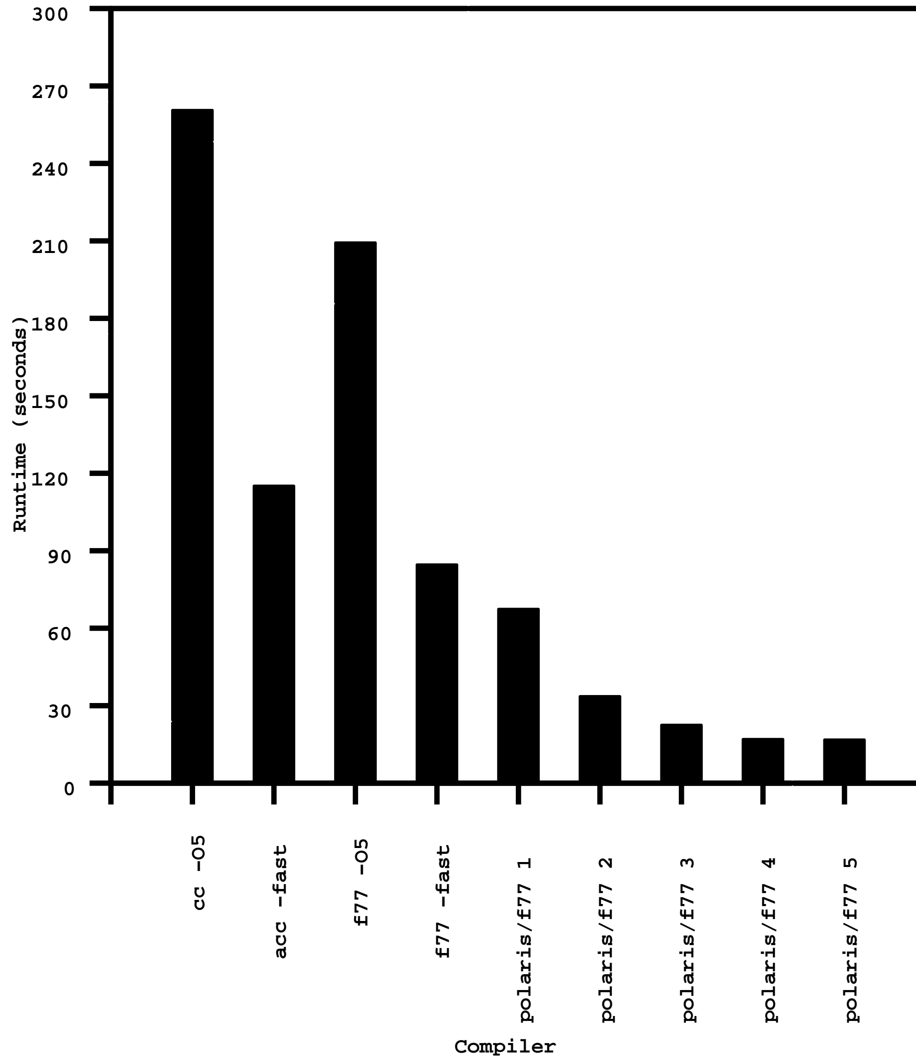
- The pow() function can be transcribed as the native FORTRAN77 ** operator.
- The declaration for variables fact and f can be moved out of the inner loop.
- The outer loop of calc can be parallelized.

FIGURE 8. Computation on vectors of numbers.

```
#include <math.h>

void calc(double *vec1, double *vec2, double *result)
{
   double *p1, *p2, *pr;
   int  i;

   p2=vec2;
   pr=result;

   for(p1=vec1; p1<&vec1[10000]; p1++) {
      for(i=0; i<1000; i++) {
         double fact=1.0;
         int f;
         for(f=1; f<i; f++)
            fact=fact*f;
         *pr += (pow(*p1,0.0+i) + pow(*p2,0.0+i))/fact;
         p2++; pr++;
      }
   }
}

int main()
{
   double vec1[10000], vec2[10000], result[10000];
   double *p1, *p2, *pr;
   int i;

   i=1;
   for(p1=vec1; p1<&vec1[10000]; p1++) {
      *p1=i/20.0;
      i++;
   }

   i=1;
   for(p2=vec2; p2<&vec2[10000]; p2++) {
      *p2=i/5.0;
      i++;
   }

   calc(vec1, vec2, result);
   return 0;
}
```

We first compiled this program with the Sun WorkShop Pro C compiler (v4.2) with the best generic and architecture-specific optimization flags and ran the executables on a five processor Sun UltraEnterprise server. The results are shown in Figure 9. Neither of these compilers support automatic parallelization of pointer codes so their performance results only demonstrate scalar optimization. Next, we used Cepheus to transcribe the program to F77 and compiled the resulting program with the Sun FORTRAN compiler (v4.2) with the best generic and architecture-specific flags and ran the executables. Finally, we compiled the FORTRAN code with Polaris in order to generate explicit Sun FORTRAN directives and then compiled the resulting code with the Sun FORTRAN compiler. The results for Polaris show the use of between 1 and 5 processors.

FIGURE 9. Performance results using different forms of compilation.



Even without using the Polaris parallelizer, the FORTRAN compiler does better than the C compiler at scalar optimization. Polaris recognizes the available parallelism and is able to describe it explicitly, allowing the Sun FORTRAN compiler to further optimize the application. The five-processor trial represents an almost seven-fold improvement in runtime over the best possible C-compiled trial.

6.0 Conclusion

By being able to reduce ANSI-C code to FORTRAN77 with Cepheus, we have demonstrated that existing parallelizing compiler technology can be applied to new or disparate languages by means of transcription. In many ways, we can even obtain greater optimization by isolating transformations to the transcriber. Control-flow normalization and pointer analysis are examples where the transcriber can express constructs that are difficult to optimize in a manner that facilitates their

parallelization. Furthermore, the task of creating the transcriber was accomplished with considerably less complexity than building a new front-end for an existing compiler; giving us tangible evidence that it is not always necessary to re-invent the compiler to achieve new types of optimization.

The Polaris parallelizer is central to the effort of improving the parallelizability of the transcribed C code for several reasons. Primarily, pointer alias analysis is only facilitated by Cepheus; we still need a second compiler with the ability to perform interprocedural array region analysis to make it work. Similarly, Polaris is used to identify and explicitly express the available loop parallelism made available by Cepheus' control-flow normalization.

We are presently working on improving Cepheus to the point where it is able to analyze and transcribe large applications and also investigating the possibility of transcription of other languages into FORTRAN. Furthermore, while language transcription at the source level is an adequate means for our studies, the ultimate goal is to express C programs in the Polaris intermediate representation directly. In doing so, we will keep modifications to the IR as small as possible. Extensions will only be made where the Cepheus transcription is unable to generate a program that can be successfully parallelized by Polaris. The current paper has presented an initial study to address this question. Extensive experimentation to give more quantitative answers is the next step.

References

[ANS93] Restricted Pointers in C. Numerical C Extensions Group / ANSI X3J11/94-019, Aliasing Subcommittee, June 1993.

[Amm92] A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering*, Volume 18(3), pages 237-251, 1992.

[BAM+96] Compiler-Directed Page Coloring for Multiprocessors. E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum and M. S. Lam. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[BDE+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Advanced program restructuring for high-performance computers with Polaris. *IEEE Computer*, December 1996.

[Coo97] Doug Cook. Performance Implications of Pointer Aliasing (Whitepaper). Silicon Graphics, Inc. http://reality.sgi.com/cook/audio.apps/dev/aliasing.html August, 1997.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296-310, 1990.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[FWH92] Friedman, Wand, and Haynes. *Essentials of Programming Languages*. McGraw-Hill, 1992.

[GH95] Rakesh Ghiya and Laurie J. Hendren. Connection Analysis: A Practical Interprocedural Heap Analysis for C. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[Ope97] OpenMP. OpenMP: A Proposed Industry Standard API for Shared Memory Programming (Whitepaper). http://www.openmp.org/openmp/mp-documents/paper/paper.html. November 1997.

[Par96] Terence Parr, ANTLR 2.00 Specification, http://www.magelang.com/antlr/ October 1996.

[PVE96] Insung Park, Michael J. Voss, and Rudolf Eigenmann. On the Machine-Independent Target Language for Parallelizing Compilers. *Proceedings of the Sixth Workshop on Compilers for Parallel Computers (CPC '96)*, Aachen, Germany, December 1996.

[Sun96] Sun Microsystems, Inc., *FORTRAN 4.0 User's Guide*, 1996.