CAN ADA CODE PERFORM AS WELL AS FORTRAN CODE?

Wendy J. Hudson
Concurrent Computer Corporation
Tinton Falls, New Jersey 07724

ABSTRACT

The coding of new training systems in Ada raises serious questions about the performance of Ada code as compared to training systems previously written in FORTRAN. When both the Ada compiler and the FORTRAN compiler are using similar methods of code optimization, the Ada code can perform as well as FORTRAN code. However, the Ada programmer must be careful in the selection of Ada features used in the system. The Ada language prevents the compiler from doing some optimizations and some of the Ada features are expensive at run-time. Examples of these features are given and benchmark results are used to substantiate the conclusions. The compilation system can also provide options and tools which can be used to fine tune the system performance. The Ada benefits of easy integration, maintainability, and reliability can be achieved without sacrificing performance.

INTRODUCTION

Run-time performance can be affected by a number of factors. The compiler used and the features of the language used will affect performance. At Concurrent Computer Corporation, we have had experience in analyzing the run-time performance of both standard and customer benchmarks written in both Ada and FORTRAN. This paper details some of the observations that we have made from performance comparisons.

Before we discuss actual results, it is important to look at some compiler techniques and language facilities that will affect performance and the pitfalls the user will encounter in trying to make direct comparisons. The paper discusses compiler optimization techniques that are currently in use in FORTRAN that may differ from Ada, difficulties in optimizing Ada, special Ada features to improve run-time performance, custom Ada features to improve performance, and benchmark results.

CODE QUALITY OF FORTRAN COMPILERS

It must first be noted that all FORTRAN compilers are not equal. Many vendors have optimizing compilers, but they all do not perform the same level of optimization. The level of code optimization varies widely. Some FORTRAN compilers perform no optimization. Some compilers do code optimization in basic blocks. Basic blocks are sections of code between statements which can cause a branch to occur (eg. IF statements, CALL statements.) There are some FORTRAN compilers which optimize code over the whole subprogram. This form of optimization is usually called global optimization. A few FORTRAN compilers have the ability to expand a subprogram inline. In this case, a call to a subprogram is essentially replaced by the code for that subprogram. A global optimizer can then optimize over the entire new routine which includes the expanded subprogram code.

The following FORTRAN code can be used to illustrate the optimization levels:

```
.
.
.
J = 1
IF (K.EQ.L) THEN
  M = J + K
  N = (J + K) * 2
  CALL SUB1 (M, J)
  N = M + N
ENDIF
.
.
.
SUBROUTINE SUB1 (M, J)
IF (J.EQ.1) THEN
  M = M + 1
ENDIF
RETURN
END
```

The following are examples of some optimizations which may be done by compilers with the indicated level of optimization:

Using a block optimizer -

- The expression J + K is evaluated only once within the IF statement.

Using a global optimizer -

- The instances of J are replaced by the value 1.
- The expression 1 + K is evaluated only once.
- The compiler can choose a faster instruction for the 1 + K calculation.

Using inline expansion and a global optimizer -

- The code for the subroutine is expanded inline. The CALL overhead is eliminated.

- The "IF (J.EQ.1)" statement is eliminated since the optimizer knows that the statement is always true.
- All the block and global optimizations mentioned above are done.
- The variable M can be eliminated if it is not used anywhere else in the program.

These different levels of optimization can produce very wide ranging run-time performance results. See Table 2 which shows results using different levels of FORTRAN optimization on a flight simulation benchmark.

Ada compilers are generally not yet mature in that they do not contain extensive levels of optimization. In addition, there are optimizations which are used in FORTRAN which cannot be done in Ada.

DIFFICULTIES IN OPTIMIZING ADA

Certainly many of the optimizations that are done by optimizing FORTRAN compilers can be applied to Ada, but the Ada standard specifically prohibits certain code optimizations. In some of these cases, the Ada language provides alternative statements which can be better optimized by the compiler.

Language Reference Manual Induced Inefficiencies:

EXAMPLE 1:

An example of this situation is the short-circuit analysis of a compound IF statement. An Ada language IF statement can use the following form:

```
IF expression-1 AND expression-2 THEN
```

The Ada standard states that both expressions are evaluated even if the first expression is FALSE. The second expression may contain function calls which have side effects.

An optimizing FORTRAN compiler usually breaks apart this statement and only permits the second expression to be evaluated if the first expression evaluates to TRUE. This can be a significant improvement in run-time performance if the statement is executed often, the first expression is usually FALSE, or the second expression is expensive to evaluate at run-time.

In this case, the Ada standard does provide an alternate form of the IF statement:

```
IF expression-1 AND THEN expression-2
THEN
```

This form of the statement would generate more efficient code, similar to the code generated by the FORTRAN optimizing compiler. However, a programmer who was originally trained to program in FORTRAN would probably use the more inefficient form of the Ada IF statement not realizing the restrictions placed on the optimizer when using that form. This is an example where some further Ada language training will be needed for the programmers in order to use the language more effectively to obtain improved run-time performance.

EXAMPLE 2:

The Ada language standard states that involution of a number (x) with a positive exponent (y) must be equivalent to repeated multiplication -- x times itself y times. In many cases, this is not the most efficient way to generate the code. A FORTRAN compiler with a smart code generator, can generate many different sequences of code for such statements. As an example, if the integer power is a power of 2, then on many machines it is much more efficient for the compiler to generate a sequence of multiply instructions that successively square the base. The Ada compiler is not permitted this flexibility. An example of better code generation is:

```
If y = 4   FORTRAN generates:   temp = x * x
                                result = temp * temp

           Ada generates:       temp = x * x
                                temp = temp * x
                                result = temp * x
```

If the program being measured contains large numbers of these expressions, then the run-time performance can be slower.

As you can see from these two examples, Ada compilers are prohibited from performing complete optimization. It becomes the responsibility of the programmer to understand the restrictions and use other techniques to achieve performance.

Expensive and Inappropriate Ada Features:

Ada has features not available to the FORTRAN programmer, that if used, prove costly to run-time performance. In these cases, comparisons to FORTRAN are not only invalid, but impossible. The programmers have to be well trained in order to understand that even though such features may provide flexibility and maintainable code, their use impacts run-time performance. The programmer may decide to use these features in areas of the program where run-time performance is not critical.

EXAMPLE 1:

One example of such a feature is Ada tasking. This allows the programmer to separate pieces of Ada code into Ada tasks. Program control can then be switched between these Ada tasks using an Ada concept called rendezvous which provides intertask communication. In some implementations, the Ada tasks can be given priorities and can be time sliced. This Ada feature is very powerful. This feature makes it very easy to implement some algorithms, however, it is generally expensive at run-time. It is perfectly appropriate to compare one Ada tasking model against another, but to compare an Ada program using tasking against a FORTRAN program may accomplish very little.

EXAMPLE 2:

Another example where Ada is expensive at run-time is in the places where Ada allows some decisions in the code to be delayed until run-time. Ada permits a great deal of flexibility and sometimes this flexibility results in run-time performance penalties.

The following Ada code is used to illustrate this case:

Case 1:

```
procedure XYZ is
  X := INTEGER := 10;
begin
  .
  .
  .
  := X + FUNC(X);
  .
  .
  .
end XYZ;
```

Case 2:

```
package XYZ_CONSTANTS is
  X := INTEGER := 10;
end XYZ_CONSTANTS;

with XYZ_CONSTANTS; use XYZ_CONSTANTS;
procedure XYZ is
begin
  .
  .
  := X + FUNC (X);
  .
  .
end XYZ;
```

In Case 1, the variable X is initialized to 10 every time that the routine is called. If X is a constant value that never changes, then this code has a run-time penalty. In Case 2, the variable X is initialized only once. The run-time performance can be significantly improved using the second approach if the routine is called a great many times.

Language Reference Manual Induced Efficiencies:

The Ada language specifies directives to the compiler which can affect run-time performance. In the Ada language, these directives are called *PRAGMAs. The defined PRAGMAs which affect* run-time performance are:

- PRAGMA OPTIMIZE,
- PRAGMA INLINE, and
- PRAGMA SUPPRESS

PRAGMA OPTIMIZE instructs the compiler to optimize the code for space or optimize for performance. *In most compilers, however, this* directive provides for all or no optimization. The use of this PRAGMA assumes some optimization technology was built into the compiler. A compiler with poor or limited optimization techniques will provide little or no benefit.

One of the hallmarks of Ada is the recognition of the technique of inlining routines to improve performance. Concurrent Computer Corporation has utilized this technique for a number of years in FORTRAN for a substantial performance improvement over non-inlined code. Inlining eliminates the save/restore cycle during the call *while also permitting appropriate variables to* remain in high-speed registers for use in the homogeneous piece of code. If the optimization capabilities of the compiler are advanced, then after the code for the routine is inlined, the compiler will have greater opportunities for optimization. Given a compiler with good optimization technology, the use of PRAGMA INLINE can result in significant run-time performance improvement.

One of the strengths of Ada and the weaknesses of FORTRAN is the subscript checking. There are no checks generated for subscript checking in the following FORTRAN code:

```
SUBROUTINE SUB1 (INDEX1)
DIMENSION ARRAY1(10)
X = ARRAY1 (INDEX1)
END
```

If the array index, INDEX1, is outside the range 1 to 10, then no error may be generated at run-time. This situation has always been the source of many bugs in FORTRAN programming.

The Ada language requires that the compiler generate a number of run-time checks in the code. These checks are used to detect range errors, subscript errors, and overflows. The difficulty is that this checking induces run-time overhead. The Ada language permits these checks to be removed by using PRAGMA SUPPRESS. These checks would not have been generated by a FORTRAN compiler, therefore, in order to compare FORTRAN and Ada performance, these checks should be turned off. The checks are very useful during the development of the system, but they can significantly affect the run-time performance.

Specialized Ada PRAGMAs:

The compiler vendor is permitted to implement additional PRAGMAs. These additional PRAGMAs are not allowed to change how the program functions, but they are permitted to generate more efficient code based on directives from the programmer. Depending on the computer architecture and the implementation of the compiler, such PRAGMAs can improve run-time performance significantly.

An example of such a PRAGMA would involve the checking of the stack at run-time. When the program enters a new routine, it must allocate space on the stack and check to see if the new allocation exceeds the stack limit. This check of the stack limit can be very expensive at run-time. If the programmer has been able to calculate the stack requirements and set the initial stack at the required amount, then the stack check is not needed at run-time. At Concurrent Computer Corporation, we have implemented PRAGMA STACK_CHECK to eliminate the generation of the stack checking code. On some benchmarks, we have seen a 10-15% improvement in run-time performance when using this PRAGMA.

The elimination of this stack check is similar to the elimination of those checks removed by PRAGMA SUPPRESS. In both cases, it is a good idea to have the checks in the code during development. However, at production time, the programmer has debugged the code and is ready to run without these checks.

Ada performance tools:

Since Ada has many new features and options, and programmers by and large are unfamiliar with the cost of these features, performance tuning tools become essential. In a benchmark world, pinpointing performance problems can be very difficult. It is much more difficult with the new Ada systems which are often very large and complex.

Consider the following situation. An Ada system has about 500,000 lines of code. The code is written in Ada and may contain some FORTRAN and assembly language routines. There are about 3,000 routines in the system. The performance is not what was expected. The compiler used implements PRAGMA INLINE and a number of other performance PRAGMAs, but they have not been used yet. The question becomes - How does the programmer determine where the performance bottlenecks are and which routines should be inlined? The programmer requires a tool to locate the slow areas in performance. A performance analysis tool can determine which routine is called the most and where the calls are generated. This routine then becomes an ideal candidate for using PRAGMA INLINE effectively.

Essentially, there should be a performance tool or evaluation technique for each performance improvement feature available with the compiler. If there is a special PRAGMA to remove the stack check, then there should be a tool to analyze the stack usage during run-time. If the compiler implements PRAGMA SUPPRESS, then the compiler should indicate how much of the generated code is checking code.

Summary of Basic Language Differences:

To summarize what has been discussed thus far, optimization technology is different between the mature FORTRAN world and the emerging Ada world. Mandated compiler techniques and facilities can both hurt and help, and therefore make it difficult to make a direct comparison. In spite of this, a compiler environment can be defined which allows a comparison on as equal a basis as possible and then actual benchmark data can be obtained.

BENCHMARK

Since Ada and FORTRAN are such different languages, it is difficult to accurately compare the code and find reasonable benchmarks. All the factors outlined in this paper can affect the run-time. The timings taken in this example try to use the same environment for all the measurements. The same machine was used and the same system parameters were used when measuring both the Ada and FORTRAN versions.

The example uses a flight simulation benchmark which was originally written in FORTRAN and then later translated to Ada. The Ada version of the benchmark contains about 25,000 lines of code in about 300 units. The FORTRAN version contains about 36,000 lines of code in 170 subroutines. The Ada version is a translation from FORTRAN and does not use many unusual Ada features.

The translation from FORTRAN to Ada resulted in some initial difficulties with the benchmark. Table 1 shows the initial run of the benchmark.

Table 1: Benchmark translated from FORTRAN to Ada:

Initial Ada run:	24.5 seconds
Ada run after minor modifications:	15.1 seconds

The performance analysis tool was used to locate the problem areas. A number of Ada coding mistakes had been made in the translation process. The most frequently called routine had a compound IF statement where the second expression was very expensive to evaluate. A simple change in the form of the IF statement used caused the run-time performance to improve 10%. Other inefficiencies detailed in this paper were detected and resolved. All of these minor modifications were resolved quickly and the run-time performance improved dramatically.

Our experience with this benchmark indicates that the programmer should be very careful when translating code from FORTRAN to Ada. Also, the programmer needs to be well trained in order to use the Ada language effectively.

The following tables compare the FORTRAN and Ada results of the benchmark after further benchmark modifications:

Table 2: FORTRAN version of the benchmark:

No optimization:	12.1 seconds
Block optimization:	8.8 seconds
Full global optimization:	7.4 seconds

Table 3: Ada version of the benchmark:

No optimization:	20.8 seconds
Block optimization:	16.4 seconds
PRAGMA SUPPRESS w/ Block optimization:	12.8 seconds
PRAGMA SUPPRESS, PRAGMA INLINE w/ Block optimization:	9.0 seconds
PRAGMA SUPPRESS, PRAGMA INLINE w/ some global optimization:	8.4 seconds

The benchmarks show that the Ada and FORTRAN performance using the same level of optimization causes very close run-time results. The FORTRAN result using block optimization (8.8 seconds) and the Ada result which uses PRAGMA SUPPRESS, PRAGMA INLINE, and block optimization (9.0 seconds) are the figures to compare in this case. In the Ada version, a few small routines used PRAGMA INLINE. The code which matched these Ada routines was not separate subroutines in the FORTRAN version.

An initial look at the results might lead the reader to compare the "no optimization" and "block optimization" results for both FORTRAN and Ada. The conclusion would then be that the Ada code was twice as slow as the FORTRAN code. However, this conclusion would be completely wrong. Both of these figures in Ada include a great deal of checking code and the call overheads of the routines which are already inlined in FORTRAN.

The Ada compiler optimizer is not as technically advanced as the global optimizer in the Concurrent Computer Corporation FORTRAN compiler. It is expected that with similar global optimization technology, then the Ada result would be equal to or better than the FORTRAN run-time performance.

When evaluating the run-time performance of Ada code, it is a good idea to also look at the standard Ada performance benchmarks which are available. The ACM Special Interest Group on Ada (SIGAda) has a working group which is called the Performance Issues Working Group (PIWG). They have developed a set of tests which test compile-time, run-time, and selected features. There is also an extensive test suite called the Ada Compiler Evaluation Capability (ACEC) tests which evaluate many specific performance characteristics. These tests are not related to the performance of FORTRAN code, but they are a good measure of one Ada compiler against another.

CONCLUSIONS

When evaluating Ada compilers for their performance characteristics, the following items should be considered:

- Look for good performance on the standard benchmarks such as PIWG and ACEC.
- Look for advanced optimization technology in the compiler.
- Look for special PRAGMAs implemented by the compiler to increase run-time performance.
- Evaluate the performance analysis tools which are available with the compiler.

In an Ada development environment, run-time performance can be improved if the following items are considered:

- Look for training courses that teach the Ada language and the run-time performance impacts of its features.
- Use the most efficient form of the Ada language statements.
- Avoid the use of the "expensive" Ada language features.
- Use the standard and special PRAGMAs available to increase run-time performance.
- Use performance tuning tools to locate slow spots in the Ada code.

In general, the comparison of FORTRAN and Ada run-time performance is very difficult. The languages are very different and Ada has a great deal of flexibility. When comparing the performance of the compilers, the same levels of optimization should be evaluated. Care should be taken to evaluate Ada performance based on the application characteristics.

Ada code can perform as well as FORTRAN code if the programmer uses the performance options and tools of Ada and is knowledgeable about the run-time characteristics of Ada features.

About the author:

Ms. Hudson is Senior Manager of the Scientific and Support Languages development group at Concurrent Computer Corporation. She holds a BA and an MS degree in Computer Science from Rutgers University. She has 12 years of software development experience involving compilers, operating systems and networking. Since joining Concurrent Computer Corporation in 1979, Ms. Hudson has been involved in the development of FORTRAN and Ada compilers, and symbolic debuggers. She is currently managing the group responsible for the development of the Ada, FORTRAN and Pascal compilers.

ABOUT THE AUTHORS

MR. MICHAEL CAFFEY is a Software Engineer at Burtek specializing in the development of simulation software for aircraft systems. Through the Ada Simulator Validation Program, Mr. Caffey has performed extensive research on the use of software engineering techniques for developing real-time software in Ada. Mr. Caffey is currently performing research on software architectures for real-time systems and code complexity measuring techniques for Ada software.

MR. MARSHALL WESTERBY is the supervisor of the Electromechanical Group at Burtek, responsible for on-board aircraft systems development. Mr. Westerby holds a degree in Mechanical Engineering from the State University of New York, Binghamton. On the Ada Simulator Validation Program, Mr. Westerby functioned as Task Manager responsible for systems integration and technical advisor.