Efficiency As A Part of Sound Software Engineering: Does Ada Need C?

By Marc L. Howell and Lynn D. Stuckey, Jr.

Boeing Defense and Space Group
Missiles and Space Division
Huntsville, Alabama

ABSTRACT

As high level computer languages (e.g. FORTRAN) became the required standard for new software implementation, simulation contractors began to seek exceptions for certain high utilization procedures. The contractors protested that they simply could not meet the customer's execution efficiency requirements if the language requirement was rigidly enforced. Customers frequently agreed to a marriage of convenience mixing FORTRAN and assembly language. The resulting problems of language mix helped lead the Department of Defense to develop a next generation language as the basis for all embedded computer systems, namely Ada. Since the efficiency requirements for embedded systems are even more stringent than real time simulation, one might have expected that Ada would fulfill the real time simulation speed requirements. However, as Ada has become the required simulation language in recent years, new contractor complaints about execution speed and memory usage have arisen. Contractors have sought waivers for these systems to implement certain procedures in the C language (the next generation assembly language) to improve efficiency.

The accepted truism has been that since a low level language executes so much faster and requires less memory than high level languages, then the loss to the customer of the desired features of the high order language is worth the gain in efficiency. Does this idea equally apply to applications using Ada? Is an Ada-C marriage convenient, much less in the customer's best interest? This paper presents a contrasting experience in two software applications that have traditionally been targets for language waivers: low level interface drivers and multi-dimensional interpolations. The paper discusses the specific benefits and costs of developing such applications in Ada and in C.

INTRODUCTION

Efficiency is defined as "the fact or quality of being efficient; competency in performance; the ratio of work done or energy developed by a machine or engine, etc., to the energy supplied to it". In the computer simulation world, efficiency breaks down into two basic concepts; execution speed and memory usage. To be efficient, a program must be "fast and small". In early simulation and training systems, efficiency was everything. Computers were inherently slow and possessed little memory. The program's run-time had to compensate for these drawbacks. The solution was usually assembly language, one step above the 1's and 0's. As computers progressed so did the languages, but assembly language continued to be the old standby for efficiency. So it remained in simulation until the introduction of C. Fast and compact, C provided efficiency and some of the "creature comforts" of advanced programming languages.

A few years later Ada was introduced as the new language of choice for the Department of Defense. It was primarily targeted at replacing the more than 250 languages used in military systems and to standardize the software inventory. Ada was developed with the whole software life cycle in mind. It was to address requirements analysis, production, maintenance, and reusability. Ada was not designed to be a next generation assembly language. Ada's designers assumed that processor and compiler technology would progress to the point that execution efficiency would no longer be a major concern for software development. Unfortunately, this has not always been the case. Numerous contractors have sought to use C as an efficient alternative to Ada in specific application areas. High level languages are generally considered not well suited for interfacing at the machine level or for fast execution speeds. The question arises: is the problem with the language, the compiler, or the designer? More specifically, is the question of efficiency a hardware problem or a problem with software designers still making choices based on priorities of the past. As an answer, this paper studies the comparison of Ada and C as they were employed to develop interpolation routines and a serial I/O driver. The paper will provide a background on the software models, their development, and the results of efficiency comparisons. Also general software principals were considered in the comparison. The paper ends with a look at the future of efficiency concerns for Ada and C.

BACKGROUND

Model Choice

The candidate routines chosen for the Ada versus C comparison were interpolation routines and a serial I/O driver. Interpolation routines are used to extract operating information from a table of known values. This is done by a set of algorithms that determine a value between two known values through the assumption of linearity. These routines are used extensively in simulation and are commonly referred to as table look-ups. Their use has, in the past, been the cause of many execution efficiency problems. In many cases they must be executed hundreds of times a second, therefore, they need to be fast. In older simulations this meant that the routines were written in assembly language instead of the required application language. The test for the interpolation routines included one, two, and three dimensional data types.

The second comparison was made on a low level RS-232 serial I/O driver. As with the interpolation routines, the concern for speed of execution has driven programmers to resort to assembly language. Other important factors in I/O drivers are the needs for direct memory access and interrupts. This type of machine level interface has long been considered cumbersome or impossible in higher order languages. The drivers were designed to read and write on a serial port. The data did not have to be in ASCII.

These test cases are representative of the software applications that have historically been developed with low level languages. The two test cases selected were developed for delivery on recent contracted systems. The interpolation routines were developed in Ada for use on a flight crew trainer. The contract involved the redevelopment of existing FORTRAN code, and as such the interpolation routines were designed based on requirements and data from a earlier program. The serial driver on the other hand was developed in C as part of a control system. This control system was developed with Ada as the primary application language, while C was used for the low level I/O. In both cases, the software was developed by competent software engineers whose language of choice and expertise was the language used for development. The use of these existing software models led to a fair and more meaningful test. The fact that they were previously developed as part of a delivered product means that the software is a better representation of code actually found in industry. The original implementations are also free of any contrived problems that may have arisen from the co-development of comparison models. The development of the new routines and a closer look at the modeling details are covered in the development section of the paper.

Test Environment

The tests for efficiency were performed in an environment based on the target system for the original models. All of the code was developed on a Sun 3/260 development system running SunOS 4.0.3. The target system was a Motorola MVME-133XT (68020) running a VxWorks real-time operating system. The Ada compiler used was the VERDIX VADSWorks compiler. The C compiler was the standard C compiler delivered with Sun development systems. No physical differences existed between the two test cases for each language implementation. A synopsis of the test environment is found in Table 1.

Model	Language	Development System	Compiler	Target System
Interpolation Routines	Ada	Sun 3/260 with SunOS 4.0.3	VERDIX VADSWorks	Motorola (68020) MVME-133XT 25MHz
Interpolation Routines	C	Sun 3/260 with SunOS 4.0.3	Sun C	Motorola (68020) MVME-133XT 25MHz
RS-232 Serial I/O Driver	Ada	Sun 3/260 with SunOS 4.0.3	VERDIX VADSWorks	Motorola (68020) MVME-133XT 25MHz Z8530 Serial Controller
RS-232 Serial I/O Driver	C	Sun 3/260 with SunOS 4.0.3	Sun C	Motorola (68020) MVME-133XT 25MHz Z8530 Serial Controller

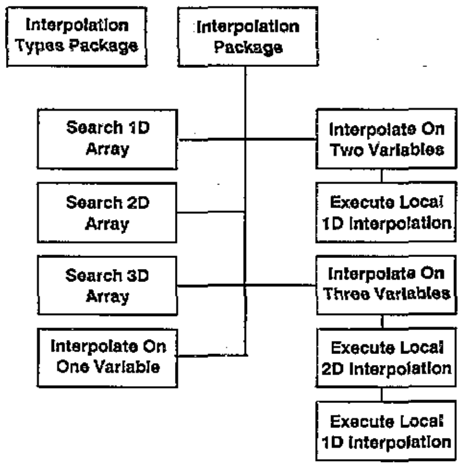
Table 1. Test Environment.

DEVELOPMENT

Interpolation Routines

The Ada interpolation routines were originally developed for use on flight trainer program. This program used the routines to determine the flight data required for the flight dynamics of a simulator. The algorithms for the interpolation routines used search techniques which were based on the last value computed. The last value was bounded by upper and lower known points. These boundaries were saved and used as the starting point for the next search, thus reducing the required search time for the next value. This approach was based on the expected application usage in flight dynamics.

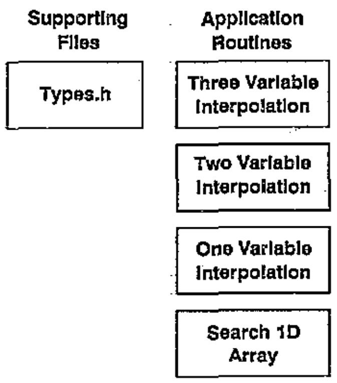
The routines were optimized by the compiler and the compiled code included calls to the floating point processor. In general, the Ada implementation of the data tables required the use of variable length arrays of up to three dimensions. It was desired to improve the calling structure of the code by using array slicing to reduce the size of the arrays being passed between routines. The Ada language has very limited capabilities for multi-dimensional array slices. This required the duplication of the interpolation routines within the two and three dimensional cases. This limitation also required longer parameter lists for the Ada routines. The code structure of the Ada implementation for the interpolation program is shown in Figure 1.



Interpolation Types Package
Interpolation Package
Search 1D Array
Search 2D Array
Search 3D Array
Interpolate On One Variable
Interpolate On Two Variables
Execute Local 1D Interpolation
Interpolate On Three Variables
Execute Local 2D Interpolation
Execute Local 1D Interpolation

Figure 1. Interpolation Routines Software Structure, Ada Version.

The C interpolation routines were developed explicitly for this comparison. They were modeled according to the same basic requirements as the Ada routines. A number of optimizations were made in the C routines which could not be used with the Ada compiler. These included the use of register allocation and array slicing into the two and three dimensional cases. Register allocation allows the C compiler to use internal CPU registers to store frequently used values whenever possible, thus significantly reducing memory access times. As a further comparison, array slicing in the C routines was developed using two approaches, first with the actual array slices and second with pointers into the arrays. No significant impact on execution was noticed. However, it was felt that passing the actual array slices was somewhat easier to follow in the code. Since array slicing was used in the C implementation, it's structure was slightly different than the Ada. Notice the apparent lack of obvious structure with the C code structure shown in Figure 2.



Supporting Files
Application Routines
Types.h
Three Variable Interpolation
Two Variable Interpolation
One Variable Interpolation
Search 1D Array

Figure 2. Interpolation Routines Software Structure, C Version.

Serial I/O Driver Routines

The C driver was written as the interface to an operator display. The required routines included a serial port initialization routine, a read port routine, a write port routine, and a routine to change the port configuration. As a standard of comparison, both implementations used the same operating system routines to achieve their purpose, i.e. VxWorks semaphores and the VxWorks I/O System interface for driver creation and calling. As with most low level I/O, it was necessary to access specific address locations in registers and memory. The C routines used declarations which defined the addresses required so that changes to the port control registers occurred whenever the value at the address changed. The bit-wise operations built into the language allowed for easy changes to these values. The read and write routines included interrupts to notify the main procedure when the respective operation was possible. The original C code was written by a programmer experienced with writing I/O drivers in C. The original code was difficult to follow due to the extensive use of abbreviated and mnemonic variable names. The structure for the C implementation of the serial driver is shown in Figure 3. In this case the code has an implied structure, as shown by the groups of logically related functions. Unfortunately, the actual code was not grouped in this manner. Even this structure is somewhat hard to follow.

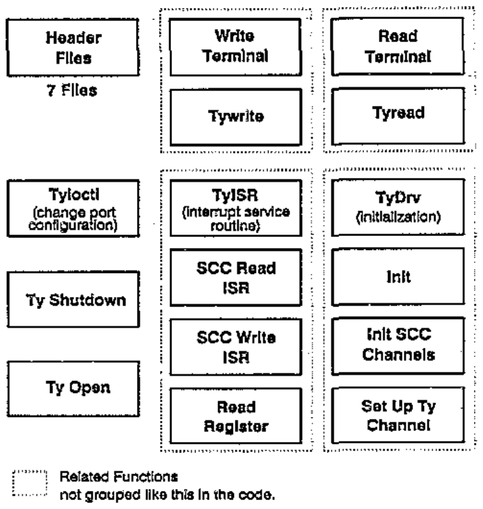


Figure 3. Serial I/O Driver Routines Software Structure, C Version.

The Ada implementation of the driver was written explicitly for this comparison. It was based on the same requirements that existed for the original C version. The program successfully used direct addressing to effect the necessary register and memory accesses. The VERDIX Ada compiler includes a set of functions which perform bit-wise operations, however, these were not used. A more desirable approach, from a portability standpoint, was to build a generic package which could be used with any integer type. This approach led to a slightly larger program size for the Ada but enhanced portability. The Ada code structure was quite different from the C. This was done as a result of the programmers experience with effective Ada packaging structure and to improve abstraction and leveling. The structure for the Ada implementation of the serial driver is shown in Figure 4.

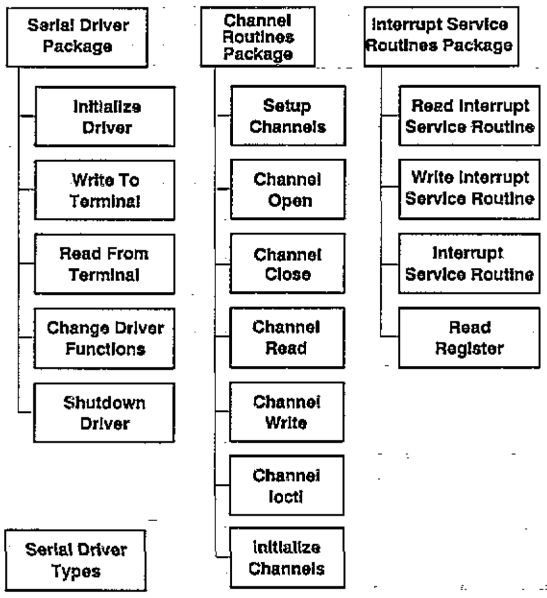


Figure 4. Serial I/O Driver Routines Software Structure, Ada Version.

RESULTS

Interpolation

The test cases for the interpolation routines consisted of several runs of different starting boundaries and desired values. The possible boundaries and the desired values included in the test are shown in Table 2. These values exercised all possible conditions for expected use of the routines. The execution times were measured using these conditions for both the Ada and C versions with the VxWorks timing function. The results were collected and ranked as worst, best, and typical (average) execution times for the one, two, and three dimensional cases. The typical case represents the most common condition for a previous value, between two known values in the table, while computing the next value, also between two known values.

Previous Boundary	Desired Value
Lower Table Limit	Lower Limit
Lower Table Limit	Interpolated
Middle of Table	Interpolated
Middle of Table	Interpolated to Next Boundary
Upper Table Limit	Upper Limit
Upper Table Limit	Interpolated

Table 2. Interpolation Test Cases.

The actual results for the one, two, and three dimensional cases are shown in Figure 5. In both language versions the best case occurred when the desired point was exactly on the upper or lower bound of the particular table searched. For this case the difference between the typical Ada and C versions is less than 5 percent. This is because the amount of code executed for this case is small and the implementations are very similar.

These internal routines acted on the entire three dimensional array with pointers to the actual desired slices. This is the primary reason for the poor performance of the three dimensional routines in comparison with C. Similar to the two dimensional case, the C version first passed a two dimensional slice and then a one dimensional slice to the respective routines. This required fewer additional parameters resulting in shorter execution times.

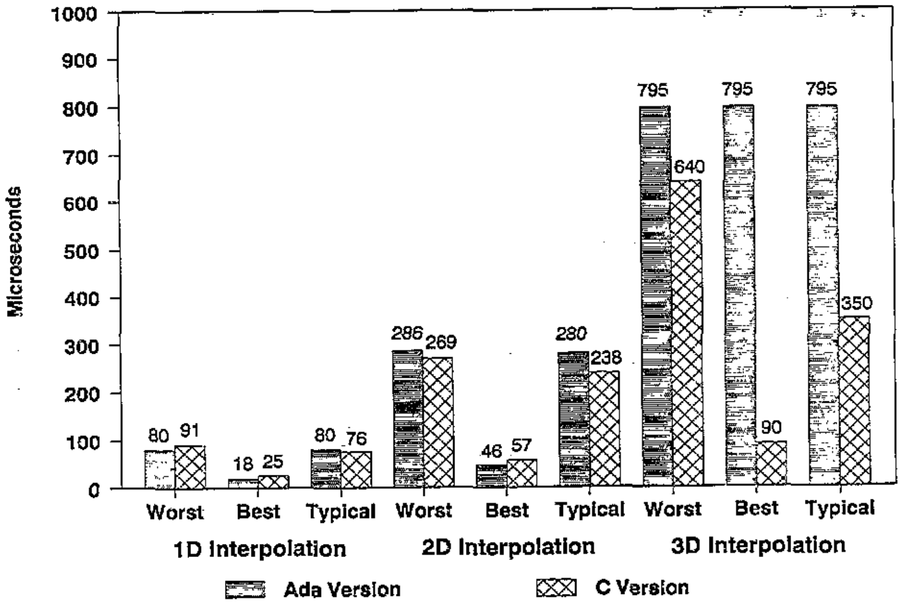


Figure 5. Interpolation Routines Speed Results.

In the two dimensional application, the variation between the two versions was greater but still fairly insignificant. For this application, the Ada version typically required 15 percent more time than the C version. This additional time in the Ada version is due to the fact that it passes the entire two dimensional array to a local one dimensional routine. It must then maintain the two dimensional indices to determine the required row from the data table. The C version uses array slicing to strip off a one dimensional slice and then passes the slice to the one dimensional routine, thus saving time and space.

For the three dimensional application, the Ada implementation required substantially more time than the C version. Again, this is a direct result of the difference between the Ada and C methods of passing array slices. The three dimensional routine used internal routines to handle the two and one dimensional interpolations it required.

The large difference in execution speed in the three dimensional case requires further discussion. Since Ada does not provide an extensive array slicing capability, the three dimensional interpolation routine could not simply call the two dimensional routine and pass it a slice from the three dimensional data array. The Ada interpolation routine used of unconstrained arrays since the size of the data array was variable. Several possible solutions to this problem were studied. The initial intuitive solution was to just declare a one dimensional unconstrained array, then declare an unconstrained of that array and so forth. The problem with this approach was that once an unconstrained array is declared it must be constrained in the declaration of the next array. Another possible solution was to declare a three dimensional array with the dimensions declared as large constants. Then an index has to be kept as to the real number of data points or the array has to be padded with zeros. This, in fact, was the method used the method used in the C version. A brute force solution was to use representation specifications to fix the memory locations internal to the three dimensional routine. Then, through the use of the Ada Unchecked_Conversion function, create the desired slices. Based on primary design criteria of maintainability, readability, and portability these approaches were not considered effective. Thus although the solution employed is slower, it is considered a better software engineering approach in the long run.

The following information represents the relative memory sizes of the interpolation routines tested. This size consists of two numbers, the total object module size and the size of the actual executable routines excluding space for data. The compiled C routines had a total size of 1764 bytes. The actual executable code required 1466 bytes which disassembled into 367 lines of assembly language instructions. The compiled Ada routines had a total size of 75916 bytes. The actual executable code required 5538 bytes, which disassembled into 970 lines of assembly language instructions. The obvious size difference in the total object modules represents the unsuppressible overhead routines which Ada automatically provides any routine to control program errors. The difference in the actual non-error program paths (assembly instructions) may seem to be significant, however, recall that the Ada version required the duplication of routines in the two and three dimensional case. The results of the interpolation module sizes are summarized in Table 3.

	Object Module Size	Executable Code(*)	Assembly Instructions
C Version	1764 bytes	1466 bytes	367
Ada Version	75916 bytes	5538 bytes	970

(*) Excluding data space

Table 3. Interpolation Routines Module Size Results.

Serial I/O Driver

The following information represents the relative execution speeds for the serial driver routines for a total of 1000 writes. This application involved communication with a serial device, a display terminal, according to the following specification. The communication process included the channel initialization, a transmission of 29 data bytes, and the channel shutdown. The channel's transmission rate was 9600 BAUD (bits per second), utilizing 1 stop bit, 8 data bits, and no parity.

For the serial driver application, the Ada implementation required 30.93 seconds, or approximately 31 milliseconds per data transmission. The C implementation required 30.316 seconds, or approximately 30 milliseconds per data transmission. These results are shown in Figure 6. Although these execution speeds are extremely close, this could be a result of the hardware limitations of the serial port or the BAUD rate rather than differences in software. If this is the case, it is interesting to note, that the Ada does not appear to limit the hardware significantly more than the C.

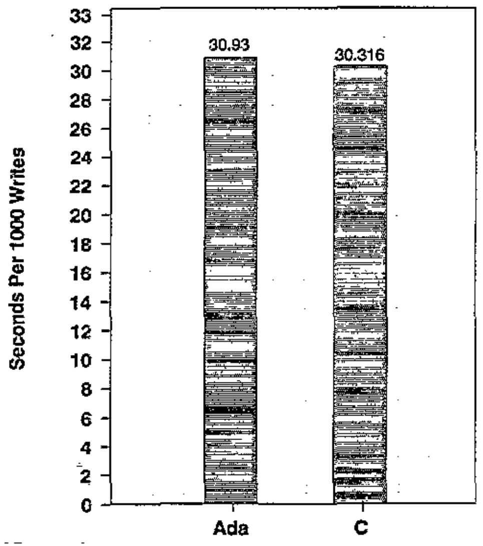


Figure 6. Serial Driver Routines Speed Results.

The following information represents the relative memory sizes of the serial driver routines. These sizes consists of the same two numbers as before, object module size and the size of the actual routines excluding space for data. The results are shown in Table 4. The compiled C routines had a total size of 4742 bytes. The actual executable code required 2174 bytes which disassembled into 560 lines of assembly language instructions. The compiled Ada routine had a total size of 62835 bytes. The actual executable code required 2380 bytes, which disassembled into 554 lines of assembly language instructions. The obvious size difference in the total object module represents the Ada overhead discussed previously. Again, notice that the differences in the actual non-error program paths is insignificant.

	Object Module Size	Executable Code(*)	Assembly Instructions
C Version	4742 bytes	2174 bytes	560
Ada Version	62835 bytes	2380 bytes	554

(*) Excluding data space

Table 4. Serial Driver Routines Module Size Results.

In general, the results from both test cases show that the use of Ada does not significantly reduce the speed efficiency of the application. The problems with array slicing in Ada are specific to the application and the design approach. The greatest drawback in the use of Ada is probably the large amount of memory required for overhead. This drawback could be a concern for applications with very limited memory.

GENERAL OBSERVATIONS

It should be noted that the routines in question were not designed with a complete emphasis on efficiency. The models were considered to be designed and coded in a manner consistent with good practices for the respective language. Software models in Ada and in C can be designed and coded similarly if not to look exactly alike. However, this would not produce representative code. The code yielded information for several other general points. Besides efficiency, the models were evaluated in terms of structure, language feature usage, maintainability, and portability. Ada is a very structured language. It is meant to encourage design and penalize poor structure. C is meant to allow quick and efficient code. C prides itself on its non-structure and its ability to "hack" a working design without regard to the quality of the layout. That is not to say that C code is not designed, but rather that it does not require or encourage sound software engineering. The main purpose of this paper was not to evaluate the software models across the entire spectrum of software engineering principles. Thus, the general observations are just that, general.

Structure and Language Usage

Several general observations were made regarding each of the languages used in this analysis. These observations had a direct effect on the methods and structures used. The most prominent ones are listed here.

[1] C does not allow multi-dimensional variable length arrays. Variable arrays are used in the Ada version of the interpolation routines to reduce the number of different sized array types needed. The C version required a different array for every case. Another possible method was to declare an array type based on some maximum size.

[2] C does not support multi-variable returns from procedures. This is due to the fact that everything in C is a function. The solution was to declare a structure which contained a variable location for the desired return values. This approach creates hidden data. The Ada version was more direct. The desired output values are listed in the procedure parameter list.

[3] The use of some C language constructs make the code less readable, for example, conditional assignment statements, extensive use of pointers '*', overloading symbols for various types, mixed type computations, and computed assignments in "if-then" statements. Such constructs are frequently used by experienced C programmers, however, usually only the person who wrote the code can easily understand what they do.

[4] Ada does not support the efficient use of array slicing. Nor does Ada support access types to multi-dimensional unconstrained arrays. C's ability to use pointers made array slicing simpler.

[5] Ada types packages were used to collect all commonly used types in the code. This feature can be very effective on large programs with many hundreds of types. C does provide a method of packaging through header files. However these are not always used efficiently.

[6] Both languages can be written to support readable and understandable code. These benefits can be enforced through coding standards. However, programmers have used mnemonic names for so long that they think mnemonics must be used for good code. This was the case with the serial driver. The C version used many mnemonics while the Ada version did not. The differences in readability were very obvious.

[7] Ada packages were also used to collect like functions and help provide leveling and abstraction. The C routines seemed to mix the functions randomly or according to calling sequence.

[8] The Ada routines could have used representation specifications and unchecked conversion to force a form of array slicing. This was considered more complex and generally 'ugly' software engineering for Ada and thus was not employeed.

Maintainability

The true cost of a simulation is only realized during the full life cycle of the software. The initial cost of development, including hardware, is minor when compared to the cost of maintaining the software. Maintainability is measured by the ease of affecting a controlled change to existing software. For these test cases, the issue of maintainability was significant for the developers of the new models. Even though the original code was less than a year old, the C serial driver had inadequate documentation and the code was almost impossible to interpret. As a result, requirements analysis for the new design of the corresponding Ada model was delayed. C code is best suited for software that is not meant to be maintained. The development effort of the Ada model was not significantly shortened by the use of the C routines. It should be made clear that the C code for the serial driver was not poorly written. It was a good working set of code that satisfied the requirements. But, the C did not encourage the design of long term maintainable code. A different story was encountered with the interpolation routines. The Ada routines provided an understandable set of requirements. The C interpolation routines, as a consequence, were simply a matter of programmer speed to develop.

Portability

Portability is the ability to transport software between computers, people, projects, and companies. Some sources refer to only the first quality as portability, and label the last three aspects 'transportability'. In the context of these tests, the edge in portability must be given to Ada. Ada is a standardized language, which in itself provides a large portion of portability that C cannot. Ada is the same from machine to machine and compiler to compiler by design. This is a problem in C, where the language might be 'normal' C, ANSI C, or even object-oriented C++. C compilers are inherently optimized for the machine on which they is hosted. This is not a bad trait for certain software products that are developed for use only on a specific machine. But in the simulation and training arena, software must be able to be adapted as the program life cycle evolves. This means that it must be able to withstand software upgrades as well as hardware upgrades. C traditionally has not directly addressed these concerns.

CONCLUSIONS

BOTH versions of the compared routines performed their required task. There were no underlying problems with language choice that degraded the operation of the routines. The C routines were on the average more efficient. This is not that surprising since C was designed with the Motorola 68000 family of processors in mind, which was the target processor in this analysis. The interesting observation is that the C routines were not significantly more efficient, except in the case of the three dimensional interpolation routines. The Ada routines provided some side benefits along the line of maintainability, portability, and readability. This is to be expected since this was a design goal of the language and a design criteria with Boeing Ada development.

To the question of whether or not Ada needs C, the answer must be 'not really'. The C code was more efficient, but not so much as to warrant the problems of requesting a language waiver, having proficient programmers (developers and maintainers) in two languages, and the added cost of two compilers and development tools. In an environment where Ada is not a mandate, then there are surely applications that would benefit from the usage of C. But if Ada is required, as it now is with all DOD programs, then these results show that the program can be implemented in that language alone and will not require a "more efficient" language (C or Assembly Language) in order to meet requirements.

The gap in efficiency is shrinking more and more as the Ada market and programming capability matures. Ada is still relatively new in terms of comparison to FORTRAN or even C. An example of new innovation has recently been developed by Tartan, Inc. Tartan has developed an Ada compiler optimized for the TI 320C30 digital signal processor. Although this is a specialized processor, Tartan was able to significantly reduce the size and execution speed of the object code compared to the C version for this processor. The results of this effort produced Ada code that was anywhere from 22% to 336% faster and with a 20% reduction in module memory size. Similar work to improve the efficiency of Ada is being done with the Intel 80960 processor which promises to improve Ada execution rates even more. These trends will help move Ada away from the stereotype of an inefficient high level software language.

REFERENCES

(1) G. Booch, *Software Engineering with Ada*, Second Edition, Benjamin/Cummings, Menlo Park, California, 1987, p. 31.

(2) Ann S. Eustice, "Tracking the Practical at TRI-Ada", Ada Information Clearinghouse Newsletter, Vol. IX, No. 1, March 1991, p. 12-16.

(3) Brian W. Kennighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition, Englewood Cliffs, New Jersey, Prentice Hall, 1988.

(4) Daniel A. Syiek and Daniel Burton, "Optimization Delivers Fast, Compact Ada Code for the TI 320C30 Digital Signal Processor", Tartan Laboratories, Inc., Monroe Pennsylvania.

(5) Tartan, Inc., "Tartan Ada Outperforms Latest C Compiler on the TI 320C30 DSP", (News Release), December 5, 1990.

(6) VERDIX Ada Development System VADS-Works, Version 5.5, for Sun-3 UNIX to Motorola 68000 family, Reference Manual, Verdix Corp., 1988.

ABOUT THE AUTHORS

Marc L. Howell is a software systems engineer with Boeing Defense and Space Group in the Missiles and Space Division. He has been responsible for software design and analysis on the Space Station program and several other Boeing projects. He is currently working on research and development studies for improving existing software implementations by using Ada software development methodologies and techniques. Mr. Howell has a Bachelor of Science degree in Electrical Engineering from the University of Alabama in Huntsville.

Lynn D. Stuckey, Jr. is a software systems engineer with Boeing Defense and Space Group in the Missiles and Space Division. He has been responsible for software design, code, test, and integration on several Boeing simulation projects including the Ada Simulator Validation Program and the Modular Simulator System. He is currently involved in research and development activities dealing with software development for weapon and threat simulators. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama in Huntsville.