Pinching Pennies While Losing Dollars

Efficiency Tradeoffs in Ada

Tony Lowe
Software Engineer
Rockwell Collins

1431 Opus Pl. Suite 200
Downers Grove, IL 60515
aalowe@collins.rockwell.com

All real time embedded software projects eventually hit the wall when they trade elegance for efficiency. The major recipient of the blame is often 'that darn tasking' or 'those silly OO concepts'. The actual culprit, unknowingly, is the front line software engineer.

Ada, being inherently different than other popular languages, has some distinctly different heuristics for creating efficient code. Even with the standardization within the Ada language, different implementations have different run time throughput and storage characteristics. If each software engineer makes small errors when declaring types, ordering statements, or adding unnecessary code, the code will likely be too slow and large, regardless of the conventional compression techniques. To successfully develop software with modern software engineering methods and in harsh real time or safety-critical environments, it is critical that each software engineer has a clear understanding of the Ada language and how their specific compiler will help or hurt the efficiency of executable software.

The purpose of this article is to challenge popular myths and highlight areas of development which can have significant performance impact for specific instances of improving efficiency. An overview of the tests in this paper:

- Fixed point versus floating point math.
- Sizing of data items and the cost on throughput.
- Using pragma inlined 'access' subprograms in place of global variables..
- Default values on data structures and types.
- Using packed data versus unpacking it.
- Using rename or inline to encapsulate functionality.
- Short-circuiting logic statements.
- Sizing of unconstrained items.
- System level testing

The tests here are similar to, and partially overlap, the Ada Compiler Evaluations System (ACES). This paper's will pull out specific issues to real time programming which have a direct effect on performance. The paper's tests are intended as a suite of evaluation criteria for Ada

elements which could have significant impact on the performance of the system for the chosen compiler, not a comprehensive compiler evaluation. The goal is to prove the language/compiler/hardware will support modern programming concepts and to point out deficiencies which could lead to adverse system performance. The idea is not to sacrifice the system design or algorithm content, but rather to optimize the usage of the language within the system.

Throughput and Sizing Testing

Since this test suite is intended to be customized and run for different compiler implementation, it is important that the tests are relatively portable and do not rely on operator knowledge of the underlying processor, operating system, or compiler. In order to provide a 'portable' testing technique, the primary method for testing throughput is to run a subprogram (which contains a test case) over several thousand or even million trials and time the overall results using the timing functions available in the Calendar package. This is not necessarily the most accurate determination of exact executable time (as opposed to counting op-codes in listing files), but it provides a technique for quick capture and analysis for comparison of different approaches. The analysis package provided with the paper, `Stop_Watch`, actually runs the test several times over different trials.

```
package Stop_Watch is

   procedure Initialize (Time_Me    : in     Timee;
                         Iterations : in     Integer := 1_000_000;
                         Steps      : in     Integer := 10);

   procedure Start;

end Stop_Watch;
```

The user initializes the package with a pointer to a parameterless procedure (type `Timee`), the subprogram under test, the total number of iterations, and the number of steps in which the iterations should be completed. The `Stop_Watch` is then started to run the subprogram under test `Iterations`/`Steps` times, and calculate the elapsed time. This is repeated `Steps` times, and the average, maximum, and minimum step duration is reported via `Text_IO`.

```
The average time of execution over      100000 trials.
steps was     106.0791 ms.
The maximum step time was     111.0229 ms.
The minimum step time was      99.9756 ms.
```

The stop watch can be initialized to run multiple times to test different scenarios, and the average time can give an indication of the required throughput for the different cases. The maximum and minimum attempt to provide an indication of Operating System 'interference' in an individual case, but are not used for analysis.

Sizing testing is performed by looking at the `'Size` of data elements and printing these values out using `Text_IO`. The size of different elements are compared to provide an indication of wasted data usage. The size of data may only have an impact on the used RAM and ROM of a system, but often can have a significant effect on the throughput of the system. If data is copied often or needs to be transmitted using peripheral devices, data structures with extraneous

information can slow down the application in
data shuffling instead of computation. By understanding the representation of data items, the developer can use or avoid particular data structures to minimize unneeded bits of data for critical data items.

Fixed Versus Float

Fixed point math is sometimes thought to be faster in its performance over floating point. This is very likely in systems without the benefit of floating point processing, but what about in systems which have the luxury? Developers could define range limited numbers to fixed points, while the larger or range-less numbers are left to floats. Is there a big difference? To analyze the differences in fixed point and floating point performance, three tests have been created. Each of the tests declare three local variables and assign the sum of two of the variables to the third. The only difference in the subprograms is the type, a float, a fixed, and an integer. The integer is added to the test to compare the fixed point math to integer math for different operations.

```
procedure Test_Type is
    A, B, C : My_Type := 9.2;
begin
    A := B + C;
end Test_Type;
```

The test is adjusted for each test by replacing the word `Type` with Integer, Fixed, or Float and the types and constants adjusted accordingly. Conventional wisdom would say that the integer and fixed point numbers would have similar execution times, with floating point numbers taking at least a bit if not significantly more time. Upon the first execution of the subprogram, however, the floating point execution was not just similar to fixed, but significantly faster! The first assumption is the floating point co-processor is significantly faster at calculations than for integers, but is this a reasonable assumption? Upon further review, the difference is actually in the declaration of the types. When declaring a fixed point type, a range and delta are required; a basic floating point declaration only requires the digits of accuracy.

```
type My_Float is digits 1;
type My_Fixed is delta 0.1 range -1_000.0..1_000.0;
type My_Integer is range Integer (My_Fixed'First)..Integer
(My_Fixed'Last);
```

Thus, the first declaration of the types had a ranged fixed and integer but an unranged float. The implication, of course, is the fixed point and integer operations add a range check, while the floating point assignment would not. When the floating point number is defined with a range constraint, the performance is essentially identical. Most applications will have a general type floating point number based upon the bits of representation, but it would seem there is a distinct throughput disadvantage to declaring a ranged float in some implementations. This brings a decision on the developers whether style (using exactly ranged numbers to represent different items), or efficiency (use general floating point numbers instead) when using floating point numbers.

This test does not confirm that fixed and floating numbers will always have similar performance, or that it is desirable for one or the other to be used in a specific scenario. The choice between fixed and floating point numbers still depends on the application needs for

calculations, safety requirements, and general data representation. This does, however, provide a few data points when selecting type representations in a system. The test did show that for my Intel processor with the Aonix compiler (7.1.1), the data types have similar performance for addition. When I changed the operation to a multiply, however, the execution time for the fixed point number jumped to 24 times that of the integer and fixed equivalent! In addition, an equivalent fixed point number would require several times larger storage size than the floating equivalent. For many systems, when the accuracy and range are small, using fixed may be a better solution. It is important to understand, though, the effects of representation of items, the fundamental types used in a system, and the size and processing required to support their definition.

Sizing of data items and the cost on throughput.

The size of data items used in a system may be designated with much thought put into the system context, but often little on the effect on the software at hand. One system we developed relies heavily on external sensors that provide all data based upon 16 bit processing. The basic types for this system all reflected the sensor's data requirements by setting the internal data sizes to 16 bits.

```
type Integer_16 is range -2**15 .. 2**15 - 1;
for Integer_16'Size use 16;
```

By creating the fundamental integer type sized to 16 bits, we are intentionally going against the grain of the processors base 32 processing. Thus each time an integer is manipulated within the system, the compiler/processor could introduce a significant impact to processing time.

To analyze the impact of sizing, two integer types are created, each sized to a range of a signed 16 bit number, but one of the types is rep claused to 16 bits and the other is left to the choice of the compiler.

```
type Integer_16 is range -2**15 .. 2**15 - 1;

type Integer_16_Sized is new Integer_16;
for Integer_16_Sized'Size use 16;
```

The initial test shows that the two types used essentially identical throughput when performing an addition of two numbers. Since the results did not meet the assumption, there must have been a problem. The actual size of the data items through the `'Size` attribute has both integers being represented with 16 bits. The next step was to size one of the integers to the 16 bit external data size, and the other to the native 32 bit processing size, though both still contain the same 16 bit range.

```
type Integer_16_Sized_32 is new Integer_16;
for Integer_16_Sized_32'Size use 32;
```

This test showed the 32 bit math to have a 30% drop in execution time over its 16 bit cousin! It would appear the ObjectAda compiler we are using chooses to size the base type to match the range definition over the native processing size, thus optimizing size over execution time.

It is vital to understand the representation of core data items and possibly use different typed data structures at different points in the system. While it may be an easier coding effort to limit

the range declarations of types to match the size of data from other devices, it is not always efficient to do so. While ideally Ada types are limited to the physical range of the represented data (e.g. `type percentage is range 0..100;`), if the type is not as large as the application's native data size, then it can be a dramatic throughput drop if the size of the item does not match the processor's native data size. A nice compromise though, especially when using data from an external source, is to create two sets of types with the same range; one type will match the external data size and the other, the internal optimal size. This may introduce some amount of overhead to copy data between variables of the different types, but the cost may have dramatic savings if data items are heavily used later in the system.

Using pragma inlined 'access' subprograms in place of global variables.

Believe it or not, the battle over the utility of global data is still raging. Global data is not likely to exist in new systems, but often in legacy systems, any attempts at data encapsulation are met with cries of inefficiency and explanations of throughput budgets. This test is intended to measure the efficiency of the compiler to access data items through subprograms in place of global variables. The concept is to replace a data item declared in a specification with two subprograms, one to read and one to write the data. The analysis of pragma Inline is performed by creating a global variable, a procedure and function to read from an encapsulated variable; and a procedure to write to the encapsulated variable and all of the above subprograms with a pragma Inline.

```
Global_Test_Data : Test_Data_Type;
-- Versus the subprograms
function Test_Data return Test_Data_Type;
procedure Put (the : Test_Data_Type);
```

For this discussion, the subject of shared variables between tasks and protected access are not being addressed only a single variable for use anywhere in the system at any time. While limiting access across different partitions is important, the focus of this test is changing access of data items away from declarations in packages to subprograms. The test performs both a 'Get' using a global variable directly, a function call (inlined and not inlined), and a procedure with an out parameter (inlined and not inlined); and a 'Put' with the global variable directly and a procedure (inlined and not inlined). The primary focus is on the 'Get', since data access is much more common than read/write capability, but the overhead of variable manipulation is important as well. The type of the variable being tested (`Test_Data_Type`) is also varied over different trials to assess the effect of data size.

This test actually was quite disappointing using ObjectAda, since the compiler does practically nothing for an 'Inlined' function call. Fortunately, the Inline of a procedure, both for 'Put' and 'Get', are identical in their execution times as to using a global variable. This testing also shows that data size can have a huge effect on variable copies, but reacts similarly between global variables and inlined calls, at least for procedures in ObjectAda. The test does point out the importance of data access times. The compiler can introduce an immense amount of overhead where it is not required, but the knowledgeable developer can both encapsulate data and have it be quick.

Default values on data structures and types.

Ada provides the ability to default a value for a record element at type declaration instead of requiring repetitive initialization at the type's usage. While this may save a few keystrokes and can prevent uninitialized usage of data, it can actually lead to 'hidden' code that can cause long initialization times and even execution times through subprogram elaboration. To investigate the extra code introduced through extraneous defaults, identical data structures are created with the only difference being the initialization of data items in one of the records.

```
type No_Default is record
   One   : Integer;
   Two   : Float;
   Three : Boolean;
   Four  : Float;
end record;

type Default is record
   One   : Integer := 0;
   Two   : Float   := 0.0;
   Three : Boolean := False;
   Four  : Float   := 0.0;
end record;
```

The two records are declared through a 'Make' subprogram which returns a record element of the given type. Upon each call of a 'Make' subprogram, a local variable of the two record types is elaborated. The defaulted record's elaboration time results in a fairly significant increase (10%) in the execution time of the 'Make' subprogram, showing the effect of initialization.

This test is not intended to ban the usage of default values in type declarations. They can be invaluable for saving debug time lost to uninitialized variables. The test does show that using default values in all record declarations is not wise for systems with hard real time requirements. To propose some basic heuristics for usage of defaults:

- If a variable element will/should always be initialized to the same value, but other items will not be initialized at creation, it is helpful to use defaults to assure the constant data items are consistently initialized.

- If the creation of a data item requires initialization upon every declaration, a default is not valuable since an assignment will soon follow elaboration. Initialization applies strongly to private elements provided through a 'Create' subprogram. Since the initialization only occurs in a single subprogram, it is easier to control the declaration of variables; thus the defaults may introduce unnecessary overhead.

- If a variable has valuable initialization values, but is also used extensively as a temporary local variable, create a constant of variable for the default initialization value. The static data items created of the type can be assigned to the default value in their declaration, and the local variables will not be forced to initialize their declarations.

Defaults are very powerful, but when real time performance is an issue, should be applied judiciously. In systems where speed is not critical, it is invariably safer to include defaults rather than not. This example is focused for the very hard real time developers who need every

last ounce of throughput available.

Using packed data versus unpacking it.

A common necessity in real time systems is to pack data for storage or communications with other systems. Whether it is an integer represented with a smaller size or packing several booleans into a word, it comes with a throughput price. Often, engineers create a record, specify the representation, forget the record is packed, and use it throughout the system. If the packed data is only used once or twice, the impact may not be significant; if the data is used repeatedly, the overhead of dereferencing packed items may be huge. Is it worth the execution time to unpack data into structures which the compiler can optimize for application usage, or is it preferable to just leave them packed? The inevitable answer to this question is largely data structure and compiler dependent. One processor may have more bit wise manipulation capabilities than another, so this issue is highly volatile. Ada uses two methods for packing data- `pragma Pack` and representation clauses. This test uses three identical records, but one is '`Pack`'ed and another is defined through rep clause. The records contain a couple of booleans and an integer ranged 0..7, or three bits when packed.

```
type Packed_Data is record
   One   : Boolean;
   Two   : Boolean;
   Three : Boolean;
   Dummy : Three_Bit_Integer;
end record;

type Packed_Data is new Not_Packed_Data;
pragma Pack (Packed_Data);
```

The test combines a logical decision on booleans and an assignment to the integer.

```
for i in 1..Times loop
   if Packed_Datas.Three then
      Packed_Datas.Dummy := 1;
   end if;
end loop;
```

The test is looped over a variable number of times to evaluate how many times a variable needs to be used to warrant unpacking. A test is also conducted to copy and use a data item. Taking advantage of the type declarations, a variable can be unpacked through a type conversion.

```
Not_Packed_Datas := Not_Packed_Data (Rep_ed_Datas);
```

The compiler will handle assignments within the packing and unpacking of data. The test is run in a loop for a variable number of times to help determine how often a data structure needs to be used to save on copying.

Conventional wisdom would say that packing data would have a significant impact on boolean evaluations since a shift and mask would need to be applied to access a single bit in a structure. Actually, Windows/Intel/ObjectAda handle boolean packing quite well, with an imperceiveable difference between unpacked and packed data. The results are quite sensible, since most processors have a mechanism for evaluation of single bits in memory as booleans. Thus for

many systems, packing booleans may have no effect on throughput, but save a lot in data consumption. On the other hand, packing integers is not necessarily as easy on throughput. As mentioned in the data sizing section, if an integer is not sized to the base size of the processor, the performance penalty can be significant. When the integer is used in the middle of the word, the throughput difference between evaluation of packed and unpacked over many trials can occasionally exceed fifty percent (50%). This partially depends on where the items are represented in the data structure. The true identifier is the copy testing since this shows the overall overhead. For this test, the copy started being more efficient between 2 and 3 loops. This may or may not be a sizable amount of calls to a single data structure depending on an application, but this also is a copy of all the data items. Some applications may only require some of the packed data items (i.e. some of the packed items are for communications protocol only and not valuable to the application), or maybe only the non-boolean elements are removed.

It is very important to understand the executable code being introduced by the compiler to handle packed data. While representation clauses allow developers to use complex interleaved data as if it were any record structure, the compiler may be adding the same functionalties the developers would be responsible for in another language. Imagine having to shift and mask in code each time you read or write a packed element. While the developers no longer need to perform these actions, the compiler still generates similar operations to make this occur. It is important to understand the usage of packed data items and potentially add functionality to pack and unpack these items instead of introducing an unknown level of overhead for the compiler to decide.

Using rename or inline to encapsulate functionality.

One architecture developed was most cleverly designed to have a single package act as the interface for an entire subsystem of code. By breaking down high-level interfaces, the entire software architecture can be described in code by looking at just a handful of packages to see the high-level functionality. From a design and analysis standpoint, the system is greatly simplified. From an implementation standpoint, a certain level of inefficiency is introduced since the 'Interface' package contained mostly subprograms whose only purpose was to call the subprogram in the appropriate package within the subsystem to perform the actual functionality. For new Ada users, the 'Interface' subprograms become subprogram calls with a single line of code, the other subprogram call. A more desirable technique for implementing these 'Interface' subprograms is to use the rename capability to redirect to the alternative subprogram.

```
procedure Without_Rename is
begin
   Do_Something;
end Without_Rename;

procedure With_Rename renames Do_Something;
```

To test the differences, three subprograms are created: the original 'pass though' call, the Ada 95 rename, and the pass though with a pragma Inline. The results are a bit surprising in that the rename is 25% faster than the double subprogram call, but the inlined subprogram call actually did very little to help the efficiency of the single call. This is possibly due to the way timing is being analyzed, but in general, a rename is a better method for completing a 'pass-though'.

The real life benefit of this would not be anywhere near as significant as 25% of the system throughput, depending on the number of inter-subsystem subprogram calls. The rename facility does, however, provide a significant tool for managing the complexity of large systems. Since most software system can be described by a set of independent functional areas, providing a single interface to the services provided helps to reduce the complexity and redundancy in a system. The subsystem may include only a few or several hundred packages to actually implement the functionality, but the 'external users' of the subsystem can limit their interactions with one or two packages. The benefits of this are tremendous when systems are large and integration efforts are huge. If several hundred packages can have their interfaces described in a single package, then other developers need only track the changes that single package. The developers of software beneath the interface can manipulate implementation software at will without any affect on the rest of the program. This method is a natural extension of the Ada specification/body implementation and synchronizes well with most modern design techniques. The rename can make the code so that you do not lose much, if any, efficiency with the use of abstraction.

Short circuiting logic statements.

One of my *favorite* generalizations in the coding standards for hard real time system is "Short circuit logic shall be used in place of the logical equivalent". It seems so broad-stroking and oversimplified, but does it help? To see the difference, two pairs of subprograms are created, each pair containing identical logic, but one uses `and`/`or` and the other, `and then`/`or else` for the logical operator. It would seem, using ObjectAda for Windows, this can have a significant gain for logical intensive systems. The `and then` and `or else` clauses provide a 50% reduction in throughput for both the short-circuited and normal state. This is for a single instance and the overall savings of short circuit logic can provide a dramatic drop in execution time in a logic-intensive system. I am not attempting to preach that all logical statements should be short-circuited. Truthfully, I have not been able to resolve for myself whether this is a good, stylistic practice or leads to confusing or misleading code. The one thing I cannot deny, it can have a significant impact on sneaking out some throughput. While a compiler option or some other method may be a preferable method to provide this savings, short circuit operations may prove very useful to finding some lost throughput in systems which are in need of a few more milliseconds.

Sizing of unconstrained items.

While size of code is not quite the issue it once was, there are occasionally throughput benefits in keeping data structures small. Often developers need to write I/O code which is both flexible and efficient. Since most I/O systems only consider the actual data being communicated as a 'bag of bits', it is easier to represent the data in a dynamically sized structure- often an unconstrained array or one hidden in a discriminated record. These structures allow different declarations of objects to have different sizes. Thus a block of data being produced by one subprogram can be of size X and a second from another subprogram can be size Y, instead of a uniform 'Max_Size' always being handled.

```
type Block_Size is range 1..106;
type Big_Array is array (Block_Size range <) of Word;
```

```
type Block (Size : Block_Size) is record
   Data : Big_Array (1..Size);
end record;

X : Block (10);
Y : Block (100);
```

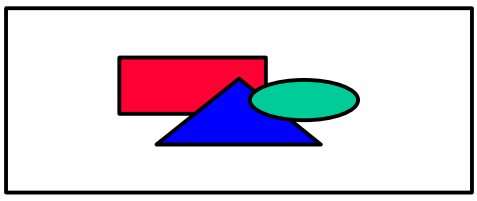
The two blocks can have dramatically different actual sizes, but can be serviced by the same subprograms. If the blocks are copied or some other operation is dependent on block size the block `X` will take only a tenth of the time that block `Y` does.

Unconstrained arrays can certainly add flexibility but can be a bit confusing to implement. So, does the compiler save enough memory usage throughout the system to be of great use? Does the compiler actually allocate the maximum size and just fill the declared size? With some easy testing, the size of the data items can be confirmed; for the compilers I have evaluated, they all optimize the size to be the declared size. The only exception to this may be when a default value is used to instantiate the discriminated record. Often, discriminated records default to the maximum size (especially with variant records) if the structure could potentially change to a different discriminant. When using a default value on a discriminated (or variant) record, make sure you understand the implication on the system size and representation.

Synchronous versus asynchronous communications.

With the addition of protected types to Ada 95, asynchronous communication can now be used along with synchronous task interactions. Is this merely a convenience or is it faster? Often developers have implemented asynchronous communications by using the synchronous rendezvous methods, so if the asynchronous method is faster, it would be a fine replacement for legacy systems. This test assumes only rendezvous' which are truly asynchronous in nature, are being replaced by the protected type equivalent. The test takes two tasks with a single 'entry' (protected or conventional) and a counter for the number of entries which occurred. The tasks are design such that each is static, thus the same physical tasks will be called each cycle. The subprogram for the tests performs the asynchronous rendezvous with the task under test, then delays for a very small amount of time to allow the task to cycle. The delay is essential since the synchronous acting as asynchronous requires that the task has completed and is ready to cycle again; the asynchronous rendezvous will occur even if the task is not completed.

My initial results were quite astonishing. The asynchronous call only took a tenth of the time to complete the same number of calls as the synchronous. Without the delay, though, the asynchronous task only rendezvous 3 or 4 out of 10,000 times. After the delay is added, the true timing differences can be seen. A bit of analysis shows the difference in execution time is due to the time taken in the call, assuming the task cycles before the delay time is completed.



If the Red box indicates the time taken by an entry call, the green by a protected entry, and the

blue is the time taken in delay, the only difference in the execution time is the difference between the green and the red; thus is the difference between the two types of rendezvous. Cycle overlap, an entry call being made before the task is ready, is measured by counting the number of times the entry is made, which must be identical for the two tests, otherwise the numbers are not quite as accurate. The protected entry seems to run about 35% faster than the entry statement. These results would seem encouraging for using protected entries over the standard entry for asynchronous rendezvous.

Analyzing critical sections of code - Looping

Sometimes, due to the nature of an application, a single subprogram or set of subprograms are called so often that they use a majority of the system throughput. On a recent project, we discovered the executable is spending a large amount of time (near 25% of the total throughput) in a single subprogram. This subprogram is called hundreds of times throughout the application and thus any inefficiency in this subprogram is multiplied by the same hundreds. By optimizing this subprogram, however slightly, we have the opportunity to save a large amount of time. In this instance, the subprogram's purpose is to copy a standard text string into the specialized format required to display the string using dedicated display hardware. When a call to the subprogram, `Assign_String`, is made, the user provides a standard Ada string, and a pointer to the data structure created to manage the specialized hardware strings. The subprogram copies the string from local memory to the pointer (located in the shared graphics hardware memory). Originally, the subprogram was originally written as a loop to copy one character at a time and increment the pointer to the graphics to the next character location.

```
for I in 0 .. This_String'Length - 1 loop
    Char_Ptr.all :=
      Graphics.Character_32Bit (Character'Pos
       (This_String (First_Char + Integer (I))));
    Char_Ptr     := Graphics.Address_To_C32Bit
      (Graphics.Dword_To_Address (Graphics.Ptr_To_Dword
        (Char_Ptr) + Ptr_Increment));
end loop;
```

Note, Ada 95 is being used so the `'Access` mechanism is not available and unchecked conversions must be used. Through listing file analysis, it was discovered the looping mechanism was introducing a large overhead (25% or more) depending on the size of the string, so a new subprogram was written to hard code the copying for all strings under a given length.

```
case This_String'Length is
...
    when 2 =
        Char_Ptr.all := Character_32Bit
          (Character'Pos (This_String (First_Char)));
        Char_Ptr     := Address_To_C32Bit
          (Dword_To_Address (Ptr_To_Dword (Char_Ptr) +
             Ptr_Increment));
        Char_Ptr.all := Character_32Bit
          (Character'Pos (This_String (First_Char + 1)));
...
```

In this case, a very specialized, maintenance intensive solution was introduced, but with a

dramatic savings on the throughput budget. I say maintenance intensive, because we actually found a discrepancy between the way the subprogram is implemented between the simulation and target environment. This caused a rewrite of each of the instances in the case statement, as opposed to a single manipulation of the loop. In this case, however, the maintenance cost does not outweigh the throughput gain. Had the specialized assignment code been applied in line with the actual code, replacing the subprogram call with the code replacement equivalent, then the effort could be days or weeks.

Sometimes a very standard coding technique introduces more execution overhead than it save in development time. No one is implying that loops should be eliminated from the language, but through good analysis, throughput hot spots can be found and partially neutralized. It is vital to pinpoint the exact area of critical code, but when performed correctly, specialized code can be a substantial savings with minimal cost on the maintainability or design of the code. I say this because, the loop was actually masking the real problem in the system, which was an inefficient compiler implementation of discriminated records. The code was thus optimized to use pointers to the character elements in place of a direct reference into the discriminant record. Even still, each bit of optimization on this subprogram is critical, but do not let the immediate solution mask what could be a greater problem. Pin point solutions are often a very powerful way to enhance the system performance.

System level testing.

While these bits can show dramatic impacts when they are isolated in simple testing, what is their impact on an entire system? The real test of these techniques is to apply them individually to a working solution and monitor the changes in code size, memory usage and throughput. I am lucky in this respect since I happened to architect a brand new Ada 95 project with a familiar target (embedded Intel) and a popular compiler (ObjectAda). The goal of this phase is to implement the above strategies one at a time, testing the throughput gains at each level to check for effectiveness. This work is ongoing and will be completed by the final publishing date.

Conclusions.

The field of software engineer is still quite diverse and non-standard, so no matter how hard the language lawyers try, they cannot get the operating system and hardware vendors of the world to agree on implementations. The only protection against the harsh real world is to have a good understanding of the language and a good set of tools to evaluate how the code is being realized on a specific target. Ideally, all of these tests will provide positive results and modern coding practices and real time deadlines can coexist. When that is not the case, these tests can hopefully provide a measuring stick for assessing which language features are problematic and which ones are scapegoats. While many optimization techniques are non-invasive, others are very complex specialized solutions. Along with the technical tradeoffs, the economic tradeoffs should also be considered. If code is extremely quick, but expensive to maintain or impossible to expand, the long reaching costs to a product could far outweigh the initial gains. Consider allocating a percentage of the throughput budget for any program to 'Modern Programming Overhead' or 'Architectural Design Overflow'. This will at least provide a limited flexibility in the software implementation and design. The goal is to understand the context of the system in which you are developing and make informed trade-offs to meet the needs of the real time and the long-term.

References

1. Ada 95 Reference Manual. International Standard ANSI/ISO/IEC-8652:1995
2. ObjectAda Compiler Documentation, V7.1.1.352 (professional edition) 1998, Aonix
3. GNAT Compiler Documentation, *I forgot the version I originally used, I will rerun the tests,* AdaCore
4. Apex NT Compiler Documentation, *I forgot the version I originally used, I will rerun the tests,* Rational Corporation
5. "A Calculated Look at Fixed-Point Arithmetic", Embedded Systems Programming, April 1998, pp. 72-76.
6. "Beware the Law of Unintended Consequences", EDN, March 18, 1999, p. 27.