C vs Ada: Arguing Performance Religion

By David Syiek
Tartan, Inc
300 Oxford Drive
Monroeville, PA 16146-2346
syiek@tartan.com

(Reprinted from *On Target*, Tartan's Compilation of Real-Time Embedded Systems News.)

Recently, I attended the Embedded Systems Conference East. Between classes, I spent time conversing with numerous other software engineers. At some point it would come out that I help build Ada/C/C++ compilers for embedded targets. The reaction to "Ada" was always interesting. Most attendees did not know much about the language, except that it was rumored to be "big and slow" and that C was felt to be much better for embedded systems programming.

I usually don't like to argue religion, but this common myth about Ada being "big and slow" is just that — a myth, possibly from early experience when compilers were not as efficient. In fact, I have found that if the language toolsets are of equal quality and maturity, benchmarks that are carefully written to be as identical as the two languages allow execute at pretty nearly the same speed. Ada versions even have a slight edge over their C counterparts!

Don't believe me? Read on...

The most recent of my many Ada vs. C run-off experiences happened this past January. I was asked to build an Ada vs. C performance case for use in an educational Ada seminar. I was to run the well-known "Hennessy" benchmarks through the Tartan Ada C3x compiler and compare the results to the ones recently computed for comparing Tartan C against Texas Instruments C in our sales literature.

The first problem was to find a version of the Hennessy benchmarks that *exactly* matched the C code. There are Ada versions of these benchmarks contained in the PIWG (Performance Issues Working Group) tests. However, on close examination, I found that these versions did not closely match their C counterparts. In the end I was forced to make my own line-for-line translation.

After a little fun verifying the correctness of the translation and even more fun duplicating the exact compiler options, linking strategy and hardware conditions used for the C tests, I managed to produce the table of ratios shown to the right.

In the table, Tartan C execution times are normalized to 1.0. Thus, slower times are greater than 1.0 and faster ones are smaller than 1.0. The geometric means of the ratios shows that Tartan Ada (v5.1) is about 9% faster than Tartan C (v2.0) which is in turn about 28% faster than TI C (v4.5).

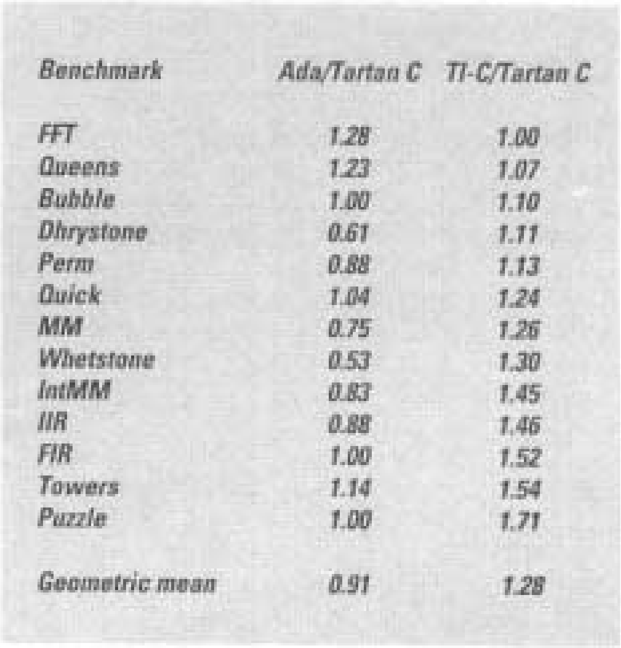
Over this set of benchmarks and with nearly identical toolsets, Ada performance was slightly better than that of C code. The comparison between Tartan C and TI C helps remove concerns that the C-specific portion of the Tartan C toolset is somehow of lesser quality than the Ada-specific portion.

I have performed many Ada vs. C performance studies over the last five years using customer application code. I assure you that these figures are representative.

Ada does seem to have a slight edge over C. I have done some research as to the reasons. Here are some of my findings.

Ada Advantage #1: Cross-Compilation Unit Optimizations

It would be foolish to build a large application as a monolith. Both the Ada and C environments provide decomposition at a number of levels. For example, both languages support the notion of separate compilation — the ability to break the application into smaller units such that each unit may be compiled separately and combined later with a linker into the final application.



Benchmark	*Ada/Tartan C*	*TI-C/Tartan C*
FFT	*1.28*	*1.00*
Queens	*1.23*	*1.07*
Bubble	*1.00*	*1.10*
Dhrystone	*0.61*	*1.11*
Perm	*0.88*	*1.13*
Quick	*1.04*	*1.24*
MM	*0.75*	*1.26*
Whetstone	*0.53*	*1.30*
IntMM	*0.83*	*1.45*
IIR	*0.88*	*1.46*
FIR	*1.00*	*1.52*
Towers	*1.14*	*1.54*
Puzzle	*1.00*	*1.71*
Geometric mean	*0.91*	*1.28*

However, there is a very key difference between C and Ada separate compilation. In C, the compilation system *never* imposes a compilation ordering. A consequence of this is that information cannot be passed from earlier compilations to later ones without the risk of creating an inconsistent system. An Ada compilation system, on the other hand, is *required* to enforce certain dependencies between compilation units and thus *must* control compilation order. Since the compilation order is under tool control, information can be saved from earlier compilations for use in later ones without fear of linking an inconsistent system.

How does Tartan Ada use this to its advantage? Here are four kinds of optimizations done by the Ada compiler using cross-compilation information:

1. Side Effect Analysis Across Compilation Units

If a compilation unit exports a `min` function that returns the lesser of the two arguments, that function likely does not have side effects. However, without proof a compiler must assume that it does. This tends to block optimizations. For example, globally visible variables may not be kept in registers across such a call. In Ada, if the `min` routine is compiled first, the compiler can determine that it does not alter any global variables and save that information for use by all callers of the `min` routine. With C, this information is lost. The C user may achieve the effect of Ada by defining `min` as a macro. However, macros are prone to problems when arguments have side effects, e.g., `min(p++,q++)`. Another solution is to include a private `min` with every compilation unit. But this wastes space. A final option is to implement some of the missed optimizations by hand around the call site. However, this introduces assumptions about the side effects of the called routine that may not hold if the routine changes in the future.

2. Automatic Inlining Across Compilation Units

In both Ada and C, it is possible to create compilation units containing libraries of small utility routines. While this is a wonderful organizational abstraction, it does have the disadvantage that for each of the functions, call overheads are incurred that can exceed the cost of the functions themselves! With Ada it is possible to keep the abstraction without losing the efficiency. The Ada compiler does this automatically by optionally expanding any routine at the call site instead of actually making the call. The Ada compilation system is allowed to do this expansion because it controls compilation order and knows that the code for the routine is consistent with the inlined version. With C, the user can get similar effects to the Ada cross-compilation through macros or through creation of private copies of routines, but as noted above, both of these methods have disadvantages and require user action.

3. Resource Usage Analysis Across Compilation Units

Across call interfaces, resources are typically divided into two classes: those preserved by the *caller* and those preserved by the *callee*. To avoid save/restore operations, compilers typically avoid all caller-preserved resources around call sites. This is suboptimal if the routine being called only uses a subset of these resources. Furthermore, when the compiler does use any caller-preserved resources, it is clearly a waste to save/restore them around a call if they are not used by the called routine. Tartan's Ada and C compilers avoid these inefficiencies by tracking routine resource usage. However, with the lack of cross-compilation information, C is unable to perform this optimization for calls between compilation units.

4. Optimal Call Site Selection Across Compilation Units

In the Tartan C3x and C40 products, there are two possible entries to every routine: one where the return address is expected to be on the stack, and the other where it is expected to be in a particular register. Either entry point may be used, but the determination of which is faster is a function of the particular routine and the conditions at the call site. Both Tartan's Ada and C compilers do this analysis when the routine being called was compiled previously within the current compilation unit, but only the Ada compiler can do this optimization properly when the routine being called is in another compilation unit.

Ada Advantage #2: C Does Not Have a True Array Type — Ada Does

With C, the *array* declaration is actually a request for memory allocation. No semantics are attached to the array object other than the base address of the block of memory allocated. Specifically, the length of the array is not automatically carried by the object either at compile-time or at run-time. With Ada, an array object carries complete length (and other) information. When the compiler is informed about the shape of an array, it may optimize more code. For example, on machines with exposed pipelines or long memory read delays, it may be beneficial for the compiler to rearrange the code:

```
if b > 0 then
   t := Table(10);
end if;
```

to become

```
RO := Table(10);
if b > 0 then
   t := R0;
end if;
```

To do this, the compiler must know that `Table(10)` is a readable location in memory even when `b` is not greater than 0. If `Table` is declared outside of this routine and passed as a parameter, then all knowledge about its size is lost to the C compiler and the optimization is blocked. With Ada this is not true. C programmers may use conventions or layers of abstraction to carry size information with memory allocations to mimic a true array type, but this will not not convey the information to the compiler and these optimizations will remain lost.

Ada Advantage #3: C Pointer Usage Reduces Compile-Time Information

Pointers are a great strength of C. However, pointer usage often reduces compile-time information and this in turn reduces optimization. For example, in the C code:

```
ap = &A; bp = &B; cp = &C;
...
for (i = 0; i < 100; i++)
    *cp++ = *ap++ + *bp++;
```

it may be hard or impossible for the compiler to determine that the pointers `ap`, `bp`, and `cp` are all pointing at non-intersecting memory. Without this information, the C compiler may be unable to perform vectorizing optimizations on the loop. Some C compilers do provide users with the ability to make assertions about pointer usage to unblock these optimizations. The burden of proof of final code correctness is (as always) on the user.

Note that the Ada equivalent of the above C loop:

```
for i in C'range loop
    C(i) := A(i) + B(i);
end loop;
```

allows the compiler to easily determine memory intersection since the objects are named directly.

Ada Advantage #4: Free Form C Loops Are Hard to Analyze

Ada `for` loops have clean compiler semantics; the iteration variable is automatically declared, is alive only in the loop, and may not be altered by code in the loop body. The iteration count is well defined. It is generally a simple task to map Ada loops to hardware-supported looping constructs.

The C `for` loop is not so well behaved. Iteration variables are user declared at any visible scope and may be alive on entry and exit of the loop. They may be altered from within the loop. The iteration count is not easy to compute given the general purpose mechanism for defining initial and continuation conditions. It takes a great deal of work for a compiler to determine that conditions are right to map a loop onto certain hardware-supported looping constructs. C users may increase the odds of success by adhering to coding standards or by hand-tuning important loops until compiler cooperation is achieved.

Conclusions

I think we can conclude that:

1. When written and compiled similarly, most Ada and C programs run equally efficiently.
2. The quality of the compiled code is determined mostly by the quality of the compiler and not of the language.
3. There are some cases where Ada code has an advantage.
4. In C the burden of optimization is often in the hands of the programmer whereas in Ada, it is automated.

It should come as no surprise that Ada provides optimization opportunities that C does not, and that it takes the burden of these optimizations off the back of the programmer. After all, the language was designed from the beginning to permit high-level programming of real-time embedded systems. The differences are not enormous, but can be significant for applications requiring the most efficient code possible.

About the Author

Dave is a graduate of Cornell University and holds a M.S. in Computer Engineering from Carnegie Mellon University. He is the senior engineer involved in Tartan's code generation technology, the primary author and maintainer of Tartan's numerical algorithms, and the primary author of some Tartan documentation. His column, Dave's Corner, is a regular feature of Tartan's On Target *newsletter.*