N89-16345

RUN-TIME IMPLEMENTATION ISSUES FOR REAL-TIME EMBEDDED ADA*

RUTH A. MAULE

Software Technology
Boeing Aerospace Company
P. O. Box 3999, M/S 82-53
Seattle, Washington 98124
Telephone: (206) 773-8607

ABSTRACT

A motivating factor in the development of Ada as the department of defense standard language was the high cost of embedded system software development. It was with embedded system requirements in mind that many of the features of the language were incorporated. Yet it is the designers of embedded systems that seem to comprise the majority of the Ada community dissatisfied with the language. There are a variety of reasons for this dissatisfaction, but many seem to be related in some way to the Ada run-time support system.

One of the more common complaints about the run-time system is that it is too big or too slow, that Ada requires excessive or unnecessary control structures to support high level language constructs that may not be used by an application. Another commom complaint is that the tasking model does not provide the type of real-time control designers are accustomed to, that the delay statement is flawed, and that rendezvous' are too expensive. These are fairly general complaints, and may well reflect real problems with the language. But there is a more fundamental problem with Ada run-time support systems of which many people are not yet aware, and that is the large number of implementation dependent characteristics which present portability problems and performance inconsistencies among validated compilers. The Ada run-time support system represents not merely a large block of additional code that must be loaded with each application, but an interface to the hardware from the source code, a real-time executive, a memory manager, and a tasking supervisor and

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

scheduler. As long as the more general Ada semantics are supported, the implementation of each of these is left largely up to the implementer. No standard interface exists between the run-time system and the application code, no consistent terminology is available for comparisons between different vendors, and no standard format defines the Ada Language Reference Manual (LRM) Appendix F, the only place where a vendor is required to describe the implementation dependent features of the system.

This paper presents some of the areas in which these inconsistencies have been found to have the greatest impact on performance from the standpoint of real-time systems. In particular, a large part of the duties of the tasking supervisor are subject to the design decisions of the implementer. These include scheduling, rendezvous, delay processing, and task activation and termination. Some of the more general issues presented include time and space efficiencies, generic expansions, memory management, pragmas, and tracing features. As validated compilers become available for bare computer targets, it is important for a designer to be aware that, at least for many real-time issues, all validated Ada compilers are not created equal.

1.0 INTRODUCTION

The high cost of mission critical embedded software was a major factor in the decision by the Department of Defense (DoD) to standardize on a single high level language (HOL). Major design decisions for this language, subsequently named Ada, were driven by the needs of embedded systems applications. Yet it is the designers of such applications that are currently among the most dissatisfied users of Ada. For many, the use of Ada is being treated with, at best, reluctant acceptance, and at worst, outright refusal.

Unfortunately, the reasons for this reaction are clear. A single language able to support the broad spectrum of DoD embedded applications must be comprehensive. Translate that to "complex," and read that "big" [Hc80], [Wb84]. For computer software, big nearly always implies "slow," and big and slow are not desirable adjectives for real-time embedded systems.

Ada is indeed comprehensive. Providing parallel processing, exception handling, and machine dependent facilities as well as structured programming support capabilities such as strong typing, modularity, readability, and generic definitions, Ada seems to have something for everyone. And early implementations have, as expected, proven to be less efficient in terms of timing and sizing than those of previous HOL's.

Then there is the policy of Ada standardization via formal validation. Where previously there was close interaction between compiler implementers and systems designers in order to develop project-specific run-time protocols and interfaces, now Ada compiler implementers must make virtually "sterile" design decisions based on the mandated necessity of passing the Ada Compiler Validation Capability test suite.

Finally, the acceptance of Ada requires the acceptance of a special, extra execution support package, the Ada run-time support system. This run-time support is required above and beyond that provided by the computer's operating executive and the application code to support Ada semantics. It appears as object code at execution time, providing many of the support functions previously designed and written by the embedded applications designer. Yet it is basically "canned" by a supplier who is unfamiliar with any project-specific needs.

The following sections are an initial view of Ada run-time support systems issues which must be defined and understood in order to make effective use of this new HOL for embedded real-time applications.

2.0 REAL-TIME ISSUES

It is to be expected that the most common complaints heard from real-time programmers about the Ada run-time support system is that it is too big and too slow. In addition, there is dissatisfaction with the tasking model, the delay statement, rendezvous costs, exception handling overhead, context switches, and more. Real-time designers, accustomed to using assembly language or HOL's not requiring additional run-time support, find it difficult to deal with the overhead that accompanies an Ada program.

A major stumbling block is presented by the Ada tasking model. Although a necessary and correctly included feature of the language, it seems that not quite enough home-work was done by the language designers to support the needs of real-time applications. Traditional approaches to real-time operating systems have relied on precise timing and tight control over the sequence and length of execution of individual system components. The Ada tasking model does not support this type of control. As pointed out by MacLaren [MI80], the cyclic executive approach, which is most commonly applied in real-time situations, is based on synchronous sequential execution, while Ada tasking is by definition asynchronous and concurrent.

Resolutions of this conflict fall into three basic categories. Some real-time designers simply refuse to use Ada tasking, and write the executive as they would have in a language that does not support tasking. The second approach is to try to force the

Ada model to fit the solution. Finally, the solution can be redesigned, making full use of the apprорriate Ada constructs. Each approach will succeed in certain situations, but the third approach is the one that must be chosen if Ada is to succeed in the long run. It is also the one least likely to be chosen, as experienced designers will be reluctant to abandon the "tried and true" cyclic executive.

3.0 AREAS OF CONCERN

The intent of this paper is to point out the areas of the Ada run-time system where implementation choices can affect real-time performance. The elements affecting run-time performance are so broad that the scope of this paper will have to be limited to some reasonable subset of them.

It is first assumed that the run-time system is written for a bare target. In this situation, the run-time system is fully responsible for run-time efficiency. It is also assumed that the target is a uniprocessor. The problems of multi-processing/multi-programming systems will be left to the more ambitious. Finally, no distinction is made as to whether the constructs discussed are actually in the run-time library or generated by the code generator. As noted before, there is no standard for this interface.

The main areas of concern seem to fall into some rather general categories, but defy rigid classification. To bring order to the discussion, the areas will be loosely grouped under the following categories: capabilities, control, kernel support, and tasking support. Capabilities and control can be looked at on either an individual level or a system level. The kernel support basically refers to the problems and elements within a program without tasking. The tasking support includes all the elements necessary to implement Ada tasking. Under kernel support and tasking support, individual system elements will be explored.

3.1 CAPABILITIES

Run-time support capabilities are those features of the run-time system that affect the application system's ability to perform its function. They represent a measure of the limitations and performance of the system with respect to what it can do and how well it is able to do it. What it can do refers to such features as support for or actual inclusion of hardware drivers, extended memory capacity, or the degree of accuracy supported for fixed or floating point calculations. How well it can do it refers to performance features such as space and time efficiency of the run-time system itself, and also of the compiler generated code. An obvious example of this is the level of code optimization achieved by the compiler. A more subtle example is the modularity of the run-time system and the user's ability to select and load only those features that are used in an application.

Specific examples of capabilities that may be necessary are numerous. The ability to specify an absolute load location for a section of code. A tracing feature that tracks the scheduler of the tasking system in real-time may be the only way to recover after a crash or to track down an elusive bug. It may be necessary to have a certain degree of accuracy for fixed point calculations. For the MIL-STD-1750A Instruction Set Architecture, the extended memory option is not a trivial addition to a run-time system.

3.2 CONTROL

Control here refers to the amount of influence possible over the system. One method of exerting control is through pragmas. Most systems for embedded targets have not fully implemented even the standard pragmas of the LRM. This will change, however, and more and more additional pragmas will be implemented as well. In this way, the vendor will be able to include special functions or capabilities not required by the standard, but of real value to a user. For example, pragmas will be able to specify that some function is not required by the application, and allow the system to eliminate that function from the resulting object code, saving at least space and quite possibly execution time.

Pragmas can also be used to request that the system do things in certain ways. Pragma optimize requests that optimization be done with one of those two often opposing goals, space and time, as the main objective. Pragma time-slice requests the system to implement a time-slicing algorithm among tasks of equal priorities. Pragma inline requests that a subprogram be included inline at each call. A pragma could be included to specify whether generic expansions should be done similarly to assembly language macro expansion, creating unique instances of code for each instantiation or if code should be shared as much as possible. Allowing the programmer to specify this information in pragmas lets the system take advantage of application-specific knowledge.

For critical real-time applications, it is useful to be able to include only those functions needed in the run-time system code that is loaded with the application. If the system is well structured, excess code could be cut from the run-time system at the source level, creating a subset system to be compiled and linked to the critical applications. One good example of this is removal of the entire tasking system. The legal aspects of this with respect to validations, however, are unknown. Another alternative is to use some type of smart linker or pre-linker that will only link in the portions of the run-time system that are necessary. This requires cooperation from the compiler, as often identification of the required run-time support routines must be done at compile time. Although the end result of these two approaches is the same, the latter one seems to be more "legal" than the former.

3.3 KERNEL SUPPORT

The implementation of the run-time kernel is of primary concern to real-time system designers. The basic support required from an Ada run-time system includes elements such as exception and interrupt handling, system clock functions, system initialization, and memory management. The effective use of registers, storage for stacks and heaps, implementation of activation records and scoping, and parameter passing mechanisms will all affect system performance.

3.3.1 SYSTEM CLOCK

The implementation of the system clock can be an important factor in the overall capabilities of the system. The counter timer chip used to drive the system clock defines the minimum granularity of time available to the system. The second level of granularity is the basic clock period, which can be found in the Ada package SYSTEM (SYSTEM.TICK) [US83]. A different level of granularity is represented by the Ada type "duration", which is not required to be the same as SYSTEM.TICK. The relationship between these values impact the system in different ways.

There is not usually a practical use for the finest granularity available from the hardware. Typically, some reasonable value is chosen for the size of the clock period, and an interrupt is generated at this rate. The interrupt handler updates the system clock, and this represents the finest resolution available. Note that, if the main processor is responsible for clock maintenance, as the resolution increases, so does the amount of time spent handling interrupts and maintaining the clock. (This is not the case if the clock is maintained independently of the CPU.)

The Ada type "duration" is not required to have the same resolution as the clock period. It is required by the Ada LRM to be at most 20 milliseconds, and is recommended that it be no more than 50 microseconds. A real-time system has timing constraints that require response within given time intervals. The clock period or the resolution of type duration must support these requirements. One system studied was found to accept a higher resolution of type duration than the system clock would permit. Although this may seem wrong at first, it was possible on that system to determine the displacement into the current clock period, and a separate timer was available for purposes of releasing the delayed tasks at the finer resolution. This is potentially more efficient, allowing finer resolution to be maintained only when necessary.

3.4 TASKING SUPPORT

The tasking supervisor typically comprises a major portion of the run-time system. In this area, many variations in implementation can appear, and can have great impact on the run-time performance of any program that includes tasking. The Ada LRM has defined the interface to the tasking system from an applications program, and a method of communication and synchronization between tasks, but has left a large part of the implementation of that system undefined. The implementer is constantly faced with a choice bewtween doing something "bare bones" quick, efficient, and simple, as would be necessary only to satisfy the LRM requirements, or going further and including features that, although not required by the LRM, are known to be highly desirable for real-time processing. If the decision is made to go beyond the requirements, then the question becomes how far to go. Added complexity will adversely affect performance, and it can be difficult to determine what is acceptable and what is not.

3.4.1 SCHEDULING

Task scheduling is an important consideration for a multitasking application. The Ada LRM does not specify a scheduling algorithm for tasks of equal priorities. Even for tasks of differing priorities, the requirementrs indicate that the task with the higher priority should be running, but the wording still leaves room for argument. Also of concern here are queueing structures, priorities, pre-emptive priority scheduling, and time slicing.

3.4.1.1 QUEUEING STRUCTURES

The order in which tasks (of equal priority) are initially put on the ready (or run) queue should be of little consequence. The Ada LRM states that programs that depend on the order of scheduling are incorrect. What is important is the method used at run time to reschedule tasks as they become ready again after a delay or some other blocked state. In theory, higher priority tasks should be dispatched before those with lower priorities. Within each priority, some fair method of sharing the processor should be implemented to prevent starvation of any single task. This may seem obvious, but the Ada LRM does not specify prevention of starvation. In fact, as long as a task does not block itself, and in the absence of synchronization through rendezvous, it is legal to allow each task to execute to completion before beginning execution of the next task. And unless some method of pre-emption is used, even a task of higher priority that becomes ready while a task of lower priority is running is allowed to wait until the currently running task relinquishes the processor.

The implementation of the queuing structures is not generally a factor in performance, but the ordering and maintenance of the queues is. The run-time system minimally provides a delay queue and a ready queue, and may additionally have a ready queue for each priority or may simply order the ready queue by priority. There may be many other queues in the system on which a task may be placed, but, (with the exception of rendezvous entry queues which will be discussed in another section) these should not affect scheduling order. The ready queue(s) should be ordered first by task priority. Within each priority, the queues should ideally be FIFO, but this is at the discretion of the implementer. The delay queue is optimally ordered by wake up time, the next task ready to wake up being at the front of the queue.

3.4.1.2 PRIORITIES

Priorities are supposed to be static except during a rendezvous, and if they are, then their effect on scheduling should be straight-forward. Some vendors may be developing some method of implementing dynamic priorities, and this will require dynamic modification of the ready queue, but as of this time, none have been announced.

Another issue in regards to priorities is whether or not higher priority tasks that become ready can pre-empt a currently running task of lower priority. This is a critical issue for many real time applications. The most common instance in which this becomes necessary is when a high priority task has been delayed for a given time span and that time span expires. The high priority task should then be allowed to pre-empt the processor from any lower priority task that may be running at the time. The alternative to this is to make the high priority task wait until the lower priority task relinquishes the processor, and then allow it to take precedence over all other ready tasks of lower priority. This is intolerable for real-time applications.

3.4.1.3 TIME SLICING

A final issue on scheduling is time slicing. Although overhead is required to implement time slicing, it is a good way to insure that each task within a priority will get an even chance at processing time. Some implementations may allow any task to be assigned an independent length of time for its time slice, or a single value may be available for modification to specify the slice length for all tasks. The user may be able to turn time slicing on and off through a software toggle. If time slicing has been implemented in conjunction with pre-emptive priority scheduling, the algorithm must take into consideration the time remaining in the slice allotted to a task that gets pre-empted so that it will be allowed to finish its slice when scheduling returns to that priority level.

3.4.2 CONTEXT SWITCHING

A terminology that is popular to toss around is the time required for a context switch between tasks. The code required and the time it takes to execute the actual context switch (that is, to change the registers, stack pointer, program counter, etc) from one task to the next are extremely hardware dependent. It is not apparent that this time has any relation to the time it takes to invoke a different task, since there is a tremendous amount of overhead involved in supporting Ada tasking that must also be done at that time. The real question is how long it takes to get a different task running once the first has given up the processor, and this reflects the amount of overhead inherent in the tasking system itself.

3.4.3 TASK TERMINATION

The part of the run-time system devoted to managing task termination can be quite extensive. The tedious bookkeeping of dependence on masters, and status of children and sibling tasks is necessary to insure that tasks terminate properly. In many cases, this overhead is necessary, but in some situations, an application may want to do away with this overhead if it does not have tasks that terminate. \This is another case where the structure of the run-time system will determine the possibility of removing or disabling this part of the system.

3.4.4 DELAY PROCESSING

When a task executes a delay statement, the run-time system must calculate the wake-up time, update the delay queue, possibly set or reset the wake-up timer, and dispatch a new task. Depending on the implementation, other functions may be required. These should be done in as efficient a manner as possible.

The processing of delay expirations can be handled in a variety of ways. If pre-emptive priority scheduling has been implemented, then a delay expiration may become a scheduling event, as the task whose delay has expired may be of a higher priority than the currently running task. To implement this, a separate timer may be set for the next scheduled delay expiration, and the code to release the task to the ready queue may be included within the handler for this interrupt. Or the code may be included within the dispatcher, and the interrupt may return to the Ada application through the dispatcher. This method forces the system to run through the scheduling routine, which may not be necessary if the released task is not of a higher priority than the running task. The code may also be included in the handler for the clock interrupt, if the resolution of type duration is equivalent to the clock period. In any case, pre-emptive scheduling requires the use of some method of regaining control of the processor by the run-time system. The efficiency with which this is accomplished is the only real consideration.

If scheduling is not pre-emptive, then the processing of a delay expiration can simply wait until the next scheduling event, whenever that occurs. Wake-up timers are not involved, and less overhead is required. But for real-time systems, this method is not really an option.

3.4.5 RENDEZVOUS COSTS

Rendezvous are effectively similar to procedure calls, yet they are much more complex to implement, and therefore create a tremendous amount of overhead for the run-time system. One task must always wait for the other to reach the point of the rendezvous, the system must invoke the rendezvous when both tasks are ready, and context switches are required between the tasks during the rendezvous. Priorities are not static during a rendezvous, and this presents additional overhead.

It is possible for the run-time system to optimize rendezvous so that the cost is more on par with that of a procedure call, but some preparation must be done at compile time. No context switch should be made if none is necessary, (such as when no code is associated with the accept statement) or when the code does not require one. This can be difficult to determine, and must be done at compile time.

4.0 CONCLUSIONS

As compilers targeted to bare computers become more common, the range of quality in run-time performance will become more apparent. Currently, many embedded systems designers are unaware of the variances simply because Ada is so new. These designers have long known what to look for in a good compiler, but many have no idea what to ask a vendor about the run-time support system.

The best run-time support for any application is determined by the individual needs of the application itself. But in a general sense, a good Ada implementation meets the basic LRM requirements, is of high and consistent quality, and is adaptable to the needs of the user. It is hoped that this paper and its successors will assist in defining the issues which impact the design and implementation of Ada run-time support systems. Once this is accomplished, ease of understanding and use should become more readily possible, allowing Ada to satisfy its requirements and original intent.

BIBLIOGRAPHY

[AR85] ARTEWG, *"Draft Charter for the Ada Runtime Environment Working Group"*, July 17, 1985.

[BR84] Baker, T.P. and G.A. Riccardi, *"Ada Tasking: From Semantics to Efficient Implementation"*, Florida State University, November, 1984.

[BS85] Baker, T.P. and G. Scallon, *"An Architecture for Real-Time Software Systems"*, Boeing Aerospace Company, Seattle, WA, July, 1985.

[GL83] Gligor, V.D. and G.L. Luchenbaugh, *"An Assessment of the Real-Time Requirements for Programming Environments and Languages"*, IEEE, 1983.

[Hc80] Hoare, C.A.R., *"The Emperor's Old Clothes"*, Communications of the ACM, Vol 24, No. 2, Feb., 1981.

[Ml80] MacLaren, L., *"Evolving Toward Ada in Real Time Systems"*, Boeing Military Airplane Company, Seattle, WA, 1980.

[US83] United States Department of Defense, *"Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A"*, Feb., 1983.

[Wb84] Wichmann, B.A., *"Is Ada Too Big? A Designer Answers the Critics"*, Communications of the ACM, Vol 27, No. 2, pp. 98-103, Feb. 1984

BIOGRAPHY

Ruth A. Maule is a systems analyst for the Ada language group of the Software Technology Division of Boeing Aerospace Company. She is currently doing research in the area of Ada run-time issues with respect to real-time embedded systems, and has written an internal Boeing document on the evaluation of Ada ru -time environments. She has done modifications to run-time systems currently in use at Boeing. She is a principal member of the SIGAda Ada Run-Time Environment Working Group (ARTEWG). She received MSCS and BSCS degrees from Florida State University in 1985 and 1983 respectively.