A QUANTITATIVE EVALUATION OF INTERRUPT HANDLING CAPABILITIES IN ADA

David G. Struble
Texas Instruments Incorporated
Dallas, Texas

Michael J. Wagner
Texas Instruments Incorporated
Dallas, Texas

ABSTRACT

Projects considering the use of Ada for embedded weapon systems have performance as their primary concern. Despite the genuine interest throughout industry to commit to Ada for reasons of portability and maintainability, the language is still viewed with reservations when being considered for use in applications with high interrupt throughput or tasking requirements. There is increased demand from users for quantitative data on Ada's ability to handle deadlines in realtime. Such data can be used as the basis for language and processor selections. This paper presents a summary of interrupt handling capabilities and quantitative interrupt benchmark measurements for three popular Ada compilers targeted to the Intel 80386 microprocessor.

INTRODUCTION

The manner in which interrupt handling is implemented varies between compilers. Some allow the user to field interrupts using standard Ada tasking mechanisms, while others implement *fast* interfaces that circumvent some of the tasking functions altogether in favor of faster, more direct control. The latter approach may involve machine code insertions or special pragmas. Where possible, this paper discusses both approaches for each compiler that offers the choice. In some cases, custom runtimes have been developed by vendors and interrupt response times through these interfaces are examined where appropriate.

To date, this report appears to be one of the first of its kind to document quantitative information on interrupt handling through Ada. In fact, in the current draft of the Software Engineering Institute's Ada Compiler Selection Handbook, the author makes the following observations:

"Interrupt handling is another feature [to be considered during compiler selection] that is important to most embedded system applications, but not to non-embedded systems. Interrupt latency and exit times as well as the functionality available in the interrupt service routine should be determined. It is difficult to include in test suites because there are many options for handling interrupts and special hardware is usually required to implement interrupts and to measure the time required to process interrupts. Interrupt handling is target architecture, compiler vendor, application, and programmer dependent."

INTERRUPT MANAGEMENT IN ADA

Determining whether or not a given Ada compiler and target processor can adequately support interrupt demands in a real-time application requires attention to several points. Guidelines discussed in [2] suggest that any successful attempt at quantifying the interrupt handling capabilities of an Ada compiler must comprehend the following:

- language support for interrupts
- availability of *fast interrupt* pragmas
- time required to get to an interrupt service routine (ISR)
- time required to return from an ISR
- deterministic nature of interrupt response times
- support for nested interrupts
- interrelationship of interrupt, hardware, and Ada task priority
- synchronization of ISR's with other Ada tasks
- support for representation specifications

BACKGROUND INFORMATION

The objective of this interrupt trade study is to quantify time intervals in an interrupt scenario using different combinations of Ada compilers (runtimes) and target processors. This quantitative information can serve as the basis for either justifying a customer's Ada requirement for a given application or showing that current compiler and runtime technology is not yet mature enough for time-critical, embedded applications of that type. The central concern among most projects continues to be Ada's ability to perform efficiently in real-time. Reports like this are intended to answer those concerns and motivate a need among compiler vendors for improvements in compiler and runtime technology across popular targets.

DEFINING INTERRUPTS TO ADA

The conventional approach to handling interrupts in Ada relies on Ada's tasking model. The Ada language includes the notion of *address clauses* for establishing the connection between an absolute hardware address and a corresponding Ada object. In the context of interrupt handling, an address clause can be used to inform the Ada compiler of the address of an interrupt service routine (ISR) to be associated with a particular interrupt. This is done by specifying the physical address of a location in an interrupt vector table or, in some cases, may specify the actual interrupt number. This location is then assigned the address of the corresponding ISR. Alternately, the Ada runtime may assign its own ISR address to a vector table entry and map to the user's ISR during execution.

As a simple example, the Intel 8086 processor maintains its interrupt vector table beginning at absolute location zero. Each entry in the table is four bytes long and consists of a 2-byte program counter offset value followed by a 2-byte segment selector number. The four bytes beginning at absolute location 24H, for example, contain the address of the ISR used to service keyboard interrupts. To make the connection between interrupt vector location 24H and an ISR named KBD_INTERRUPT, the following address clause would be used:

```
for KBD_INTERRUPT use at 16#24#;
```

The actual interpretation of an address clause is system dependent, so the expression in the for-statement could specify an offset instead of an absolute location. In the current example, the for-statement serves only to identify a location to hold the address of the actual ISR, while the ISR itself is defined in terms of Ada's tasking semantics. The following code segment shows how the address clause is associated with the corresponding entry in the task handling the interrupt:

```
task KBD_ISR is
    entry KBD_INTERRUPT;
    for KBD_INTERRUPT use at 16#24#;
end KBD_ISR;
```

The above code serves as the specification portion of an Ada task that will run in response to a specific event (i.e., a keyboard interrupt). The body of the accept statement in the task named KBD_ISR will provide the actual sequence of steps needed to handle keyboard interrupts. Assume that the keyboard interrupt handler need only accept a keystroke and place it in the next available byte of a fixed length buffer. The Ada task needed to accomplish this could be written as shown below.

```
with GBL;
with DOS;
task body KBD_ISR is
    CHAR : character;
begin
    loop
        accept KBD_INTERRUPT do
            DOS.READ_FROM_KBD ( CHAR );
            GBL.BUFF ( GBL.PTR ) := CHAR;
            GBL.PTR := GBL.PTR + 1;
            if GBL.PTR = GBL.MAX_PTR
              then GBL.BUFF_OVFLOW := TRUE;
            end if;
        end KBD_INTERRUPT;
    end loop;
end KBD_ISR;
```

Interrupts that are queued are considered as *ordinary* entry calls while interrupts that are lost if not processed immediately are considered as *conditional* entry calls. An interrupt can be executed before any scheduling is done by the compiler to improve response time, i.e., scheduling activities can be deferred until *after* the interrupt is handled.

The preceding example is oversimplified, but illustrates the basic Ada approach to implementing an interrupt handler. However, depending on the particular compiler and runtime being used, it is possible to implement ISR's more efficiently. Most Ada compiler vendors now make custom runtime interfaces available that allow real-time embedded applications to be written more efficiently and with less or no concern for the overhead implicit in the standard tasking approach.

INTERRUPT TASK CREATION AND ACTIVATION

Tasks designed to handle interrupts are *normal* Ada tasks — normal in the sense that they behave in the same manner as any other task and are subject to the same scheduling criteria. In the example of the preceding section, the task named KBD_ISR will be activated when the application program is given control. Assuming that a keyboard interrupt is not pending at startup, the KBD_ISR task will be placed on the ready queue at a priority higher than any normal Ada task (for *some* runtime implementations, task priority will not be adjusted until *after* the interrupt.) When a keyboard interrupt occurs, the Ada runtime will attempt to activate the KBD_ISR subject to the restrictions imposed by the Ada tasking model. This context switch then passes control to the interrupt service routine which runs to completion or until a higher priority interrupt preempts the currently running ISR.

It is important to make the distinction between the runtime's ISR and the Ada task associated with the interrupt. The runtime ISR will cause the Ada task to be scheduled on the ready queue. The Ada task is blocked at the accept statement waiting for someone to rendezvous with it (usually the *hardware interrupt*, although it could be another Ada task). In other words, the runtime ISR *simulates* the rendezvous with the Ada task.

This scenario applies in those cases where an ISR is designed to run within the normal framework of Ada tasking. For *fast* interrupt entries and certain custom runtime interfaces, the normal rescheduling and queue maintenance operations associated with the tasking model can be circumvented, but this is at the expense of producing very specific code that will be difficult to port to new compilers and targets. From the standpoint of efficiency, however, the time savings can be substantial: the average time to rendezvous using a normal interrupt task (Intel compiler) is 264.8 microseconds (based on 16 Mhz clock rate). Using a *fast* interrupt interface, this time drops to 21.4 microseconds.

Examples of alternative approaches to handling interrupts under different compiler configurations are presented in the sections that follow.

VENDOR-SPECIFIC APPROACHES TO INTERRUPTS

The details of implementing an interrupt handler are dependent on the target architecture and the individual compiler vendor. Nearly all of the vendors, except for Verdix, use the standard task structure for interrupt handling. An address clause is used to map a task entry to a physical interrupt. This is also dependent on the microprocessor architecture. The Intel 80386 has software tables

for the interrupt vectors and the address clause usually references an entry into this interrupt table.

Since there is runtime overhead associated with the tasking structure and the rendezvous concept, most of the compiler vendors provide a *fast* interrupt handling method. The specific details of the implementation of a fast interrupt handler vary for each vendor, but most use a pragma in the task specification to signal that a fast interrupt interface is to be generated by the compiler.

DDC-I

The DDC-I Ada compiler allows two methods of interrupt handling: the standard Ada method as defined in the LRM and a fast interrupt handler to capture interrupts more quickly. The compiler supports address clauses which allow the programmer to map the task entry to an entry in the 80386 Interrupt Descriptor Table (IDT). Both the normal interrupt method and the fast interrupt method use address clauses to assign a task as an interrupt handler. Each method has the same program structure, although the fast method requires use of the pragma INTERRUPT_HANDLER to distinguish the task from normal tasks. The *fast* interrupt method puts some restrictions on how it can be used. The *normal* interrupt method allows the task to be used the same way as any other task *and* as an interrupt handler.

DDC-I NORMAL INTERRUPT ENTRY

When using the normal interrupt entry method to handle an interrupt, the interrupt vector is mapped onto a normal conditional entry call. In this manner, the task can be used within the program environment just like any other tasks that are defined. There are only two constraints on the task structure:

- The affected entries must be defined in a task object only (not a task type)
- All entries of the task object must be single entries with no parameters

When the interrupt handler task is linked in the main program, the compiler generates the necessary vector that is entered into the 80386 IDT. This interrupt vector points to a routine in the Ada runtime that completes the necessary Ada context switch before giving control to the task entry. If the protected mode version of this compiler is used, then the user must modify a system build file to make entries into the 80386 IDT. Because this method is more flexible than the fast interrupt handling method, it takes longer to execute compared to the fast interrupt method. This is a result of having to complete an entire context switch as if it were a normal task switch. An example code section follows to illustrate the form of the task specification and body.

```
with system;
package INT_HANDLER_PACKAGE is
    task INT_HANDLER_TASK is
        entry INTERRUPT;
        for INTERRUPT use
            at (offset = 16#81#, segment = 16#0000#);
    end INT_HANDLER_TASK;
end INT_HANDLER_PACKAGE;

package body INT_HANDLER_PACKAGE is
task body INT_HANDLER_TASK is
    begin
        loop
            accept INTERRUPT do
            -- code to handle interrupt is here
            end INTERRUPT;
        end loop;
    end INT_HANDLER_TASK;
end INT_HANDLER_PACKAGE;
```

DDC-I FAST INTERRUPT ENTRY

The fast interrupt handler method is less flexible than the normal interrupt method, but because of this reduced flexibility, an increase in performance is realized. This method uses a pragma INTERRUPT_HANDLER to make the compiler transfer control directly to the accept statement in the interrupt handler task. The constraints placed on the task when using this pragma are:

- The affected entries must be defined in a task object only (not a task type)
- The pragma must appear first in the specification of the task object
- All entries of the task object must be single entries with no parameters
- The entries must not be called from any other task

The body of the task object must not contain anything other than simple accept statements (possibly in a loop) referencing only global variables, no local variables. In the statement list of a simple accept statement, it is allowed to call simple, single and parameterless entries of other tasks, but no other tasking constructs are allowed. The call to another task entry, in this case, will not lead to an immediate task context switch, but will return to the caller when complete. Once the accept is completed, the task priority rules will be obeyed and a context switch may occur.

When the pragma INTERRUPT_HANDLER is used, the 80386 IDT segment is updated at link time. The entry in the IDT will be updated to point directly to the interrupt routine generated by the compiler to make the task entry call. This leads to faster response time in handling an interrupt and shorter context switch times. If the protected mode version of the compiler is used, then the user must modify a system build file to make entries into the 80386 IDT. A short section of the code shown above is listed below to illustrate the location of the pragma INTERRUPT_HANDLER in the context of the task specification.

```
task INT_HANDLER_TASK is
    pragma INTERRUPT_HANDLER;
    -- above pragma applies to entry INTERRUPT
    entry INTERRUPT;
    for INTERRUPT use
        at (offset = 16#81#, segment = 16#0000#)
end INT_HANDLER_TASK;
```

The remainder of the package specification and body is the same as the code section listed in the normal interrupt handler section.

With the fast interrupt handling method, the compiler generates code to exit the interrupt handler and to call an end of interrupt routine. This routine checks to see if the interrupt was caused by

the timer before exiting back to the interrupted program. This extra code is not necessary when the programmer knows that the interrupt was not caused by the timer and can be eliminated by adding a machine code insertion at the end of the interrupt handler. This procedure call takes care of the normal interrupt exit housekeeping – except for the timer check. The code generated by the compiler still includes the call to the end of interrupt routine, but since the code insertion ends with an IRET instruction, the compiler generated code never gets executed. This procedure was provided by DDC-I to improve the exit time for the fast interrupt handler. It is *not* included in their normal compiler. They are planning, however, to eliminate the timer check in future releases. The following section of code shows what is necessary to accomplish this faster exit time.

```
task body INT_HANDLER_TASK is
    begin
        loop
            accept INTERRUPT do
              - - code to handle interrupt is here
              FINISH_INTERRUPT; - - extra procedure
            end INTERRUPT;
        end loop;
    end INT_HANDLER_TASK;

with machine_code; use machine_code;
procedure FINISH_INTERRUPT is
    begin
        machine_instruction'(register, m_POPD, GS);
        machine_instruction'(register, m_POPD, FS);
        machine_instruction'(register, m_POPD, ES);
        machine_instruction'(register, m_POPD, DS);
        machine_instruction'(none, m_POPA);
        machine_instruction'(none, m_IRET);
    end FINISH_INTERRUPT;
```

Performance measurements were made with and without this additional procedure. It is not included in the standard compiler package, but it can be used to improve the interrupt exit time.

TELESOFT

The Intel Ada386 compiler by TeleSoft allows two methods of interrupt handling: the standard Ada method as defined in the LRM and a fast method using a *function mapped optimization* to handle interrupts more quickly. The compiler uses address clauses to designate an interrupt entry. The address clause refers to the address of an interrupt descriptor in the 80386 Interrupt Descriptor Table (IDT) rather than the address of a physical interrupt vector. Both the standard method and the function mapped optimization method use address clauses to assign a task as an interrupt handler.

Each method has the same program structure, but the function mapped optimization method requires the use of the pragma INTERRUPT (FUNCTION_MAPPING) to distinguish the task from normal tasks. The function mapped optimization method puts some restrictions on how it can be used, while the normal method allows the task to be used the same way as any other task, as well as to be used as an interrupt handler. The compiler also makes a provision for a *synchronization optimization* which causes the handler task to become ready to execute without requiring an actual context switch as part of servicing the interrupt. This optimization is applied in both methods of interrupt handling whenever possible and does not need to be specified explicitly.

When an interrupt occurs, the runtime enters the interrupt servicing procedures with interrupts disabled. After the runtime determines if the interrupt handler is a fast interrupt, it either gives control to the fast interrupt handler or does a full task context switch. The standard and function mapped interrupt handling methods are discussed below.

TELESOFT NORMAL INTERRUPT ENTRY

When using the normal interrupt entry method to handle an interrupt, the interrupt vector is mapped onto a normal conditional entry call. In this manner, the task can be used within the program environment just like any other tasks that are defined. If the interrupt handler is not ready when an interrupt occurs, then a backup handler in a failure task is invoked instead. This failure handler can be explicitly defined just like any other interrupt handler. The normal interrupt handler does not have any restrictions regarding what is allowed in the body of an accept statement. Although it is not suggested for an interrupt handler, the code could contain entry calls to other tasks or even delay statements. In the general case, this method of handling an interrupt requires a full Ada context switch to the interrupt handler task and then a full context switch back to the interrupted task when the rendezvous is completed. A sample code section is shown below to illustrate the form of the task specification and body for both an interrupt handler and for a failure handler.

```
with INTERRUPT;
package INT_HANDLER_PACKAGE is
    FAIL_DESC : INTERRUPT.FAILURE_DESCRIPTOR;
    INT_DESC : INTERRUPT.DESCRIPTOR
        := INTERRUPT.SOURCE (16#27#, FAIL_DESC);

    task FAIL_HANDLER_TASK is
        entry UNHANDLED_INTERRUPT;
        for UNHANDLED_INTERRUPT use
            at FAIL_DESC'address;
    end FAILURE_HANDLER_TASK;

    task INT_HANDLER_TASK is
        entry INTERRUPT;
        for INTERRUPT use
            at INT_DESC'address;
    end INT_HANDLER_TASK;
end INT_HANDLER_PACKAGE;


package body INT_HANDLER_PACKAGE is
    task body FAIL_HANDLER_TASK is
    begin
        accept UNHANDLED_INTERRUPT do
            - - code for unhandled
            - - interrupts is here
        end UNHANDLED_INTERRUPT;
    end FAIL_HANDLER_TASK;
```

```
task body INT_HANDLER_TASK is
begin
    accept INTERRUPT do
        -- code to handle interrupt
        -- is here
    end INTERRUPT;
end INT_HANDLER_TASK;
end INT_HANDLER_PACKAGE;
```

TELESOFT FUNCTION MAPPED OPTIMIZATION

The function mapped optimization method is less flexible than the normal interrupt method, but because of this reduced flexibility, an increase in performance is realized. This method uses the pragma INTERRUPT (FUNCTION_MAPPING) to make the compiler transfer control directly to the interrupt handler accept statement in the interrupt handler task via a function call. During elaboration of the task, the entry made in the IDT or interrupt table is a vector that points directly to the code generated by the compiler to handle the interrupt. When using this method of handling an interrupt, there are several restrictions placed on the types of structures allowed in the task. These constraints are:

- The accept statement cannot reference any dynamically allocated variables of the task
- The accept statement cannot interact with other tasks during the rendezvous
- A priority cannot be specified for a function mapped task
- The function mapped optimization can only be used with three constructs: a simple accept statement, a while loop enclosing a single accept, or a select statement that includes an interrupt accept alternative

When the pragma INTERRUPT (FUNCTION_MAPPING) is used, the body of the accept statement is mapped into a function rather than a procedure. This is done so that the return value of the function can contain the value of the loop control expression. This value is used by the runtime to determine whether or not there will be another cycle through the loop. If the function returns a value of true, the task does not need to be rescheduled and this results in a fast exit from the interrupt handler. If the loop is completed, then rescheduling is necessary and is performed by the runtime. A short section of the code shown above is listed below to show the location of the pragma in the context of the task body.

```
task body INT_HANDLER_TASK is
begin
    pragma INTERRUPT (FUNCTION_MAPPING);
    accept INTERRUPT do
        -- code to handle interrupt is here
    end INTERRUPT;
end INT_HANDLER_TASK;
```

The package specification and the remainder of the package body are the same as the code section listed in the normal interrupt handler section.

VERDIX

The Verdix Ada compiler for 80386 targets allows only a single method of interrupt handling which is not the same as the method specified in the LRM. It also includes an extension facility using *signals* for communication between an interrupt handler and a task. The Verdix compiler has a runtime kernel that is linked separately from the user program. This kernel has several board specific support packages for configuration, interrupts, timer, and the operating system. Any interrupt handlers must be with'ed into the kernel and are elaborated at the end of the kernel startup initialization. At this time, the interrupt handlers are attached to the hardware vectors. When an interrupt occurs, the CPU vectors directly to the interrupt handler with no runtime overhead. Since there is no kernel code added to the interrupt handler, the user must be careful to save any processor information that will be modified during the interrupt. This also requires the program to restore the processor state when it is finished.

Verdix provides a subroutine that the interrupt handler can use to replace the interrupt vector at elaboration time. They also provide a shell interrupt handler which can be used as a template for the program's interrupt handler routines. The interrupt handler procedure saves the necessary machine states and then gives control to the user's interrupt handler. After the interrupt is processed, control returns to the interrupt handler program and a call is made to the runtime kernel. The kernel finishes restoring the original machine state and gives control back to the interrupted program, not to the interrupt handler procedure.

Control is passed to the interrupt handler with interrupts disabled. During execution of the handler, interrupts can be enabled and disabled with the STI and CLI instructions. The handler can also change the priority of the interrupt handler program. The kernel does not support exception handling, so any interrupt handler program must be compiled with constraint checks suppressed and should not explicitly raise an exception. If any floating point instructions are used, then the entire state of the floating point coprocessor must also be saved and restored.

The Verdix kernel also supports *signals* to allow interrupt handlers to communicate with executing Ada tasks. A signal can be created for use in an interrupt handler and then the handler can *post* the signal when it gets executed. A normal Ada task can be used to have an entry that corresponds to the interrupt's signal. When the signal is posted, the task containing the accept for that particular signal will get called and a rendezvous will occur. This task will essentially contain the code to handle the interrupt. The user's interrupt handler can either handle the interrupt completely or make use of the signal to process the interrupt.

These tasks are not limited to being used as interrupt handlers. They can be used by other parts of the program just like any other task. The only restriction placed on interrupt entries is that they may not have parameters. This method is more flexible, but takes longer to execute since after the signal is posted, the interrupt handler continues and eventually calls the procedure COMPLETE_INTERRUPT. Control is passed to the kernel and when it reschedules the tasks, it will find the signal pending and simulate a rendezvous with the associated task entry. The kernel does not provide any queuing mechanism for signals, hence if the signal is posted more than once before the rendezvous occurs, the extra interrupt events will be lost.

If any data is to be used by both the interrupt handler and the user task, a portion of memory in a fixed location must be defined to contain shared variables. The kernel does not provide a method

for protecting this data so a mechanism must be set up to prevent bad data at this area in the event that an interrupt occurs while the user task is reading or modifying the data. Verdix provides two routines, ENABLE_INTERRUPTS and DISABLE_INTERRUPTS, that the user task can use to protect itself when it is accessing the shared data. Some sample code segments are listed in Figure 2 to illustrate the use of the interrupt handler, signals, tasks and a shared data structure.

PERFORMANCE DATA

This section of the report presents timings for the various interrupt handling methods just discussed. Results are shown for DDC-I version 4.2, Intel Ada/386 version 3.20.03, and Verdix VADS version 5.5, all targeted to the 80386.

INTERRUPT TIMELINE DEFINITIONS

The interrupt response times documented in the *results* section show two values for each compiler/target combination. The first value, labeled T1, is the time measured from the start of the processor's interrupt acknowledge signal to the start of the first instruction of the interrupt service routine. This is the *interrupt latency* or period during which runtime *prologue* code is executing to prepare for handling the interrupt. When standard Ada tasking is used to implement the ISR, T1 will include the overhead involved for a context switch and any other task maintenance operations including enqueuing, updating, and dequeuing of the affected Task Control Blocks (TCB's).

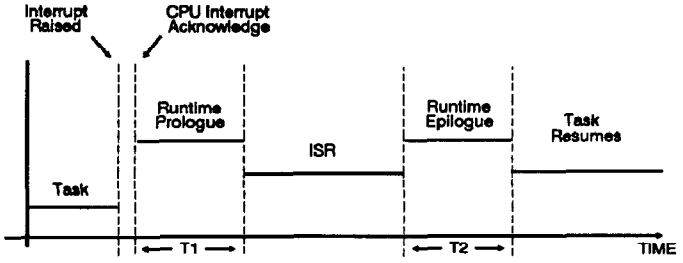

Figure 1. Interrupt Timeline.

The second value shown, T2, is the period following the interrupt service routine during which runtime *epilogue* code is executing to perform a context switch back to the originally interrupted program. Figure 1 shows the positions of T1 and T2 on the interrupt timeline. When standard Ada tasking is used to implement the ISR, T2 will include the overhead involved for another context switch and, again, any other task maintenance operations including enqueuing, updating, and dequeuing of the affected TCB's.

TEST SOFTWARE AND HARDWARE METHODOLOGY

The primary objective of writing software to test Ada interrupt handling capabilities is to produce reliable and repeatable results showing the best possible performance of each compiler. This objective was met by writing a short program that actually causes an interrupt, not by having a random interrupt occur sometime while the program is executing. The details of each program are different for each compiler, but the basic idea is the same for all of them.

With the 80386 target, a physical I/O pin was wire-wrapped to one of the interrupt input pins. The software consisted of a short main program and an Ada interrupt handling routine specific to the particular compiler. The main program had machine language insertions to toggle a pulse on the I/O pin and effectively cause an external interrupt. The interrupt handler was invoked by the runtime to handle the interrupt. When it was finished, the main program was able to complete execution.

INTEL SBC386/31 BOARD

The SBC386/31 board is a Multibus I single board computer with an Intel 80386 microprocessor and 80387 math coprocessor. The clock speed is normally 20 Mhz, but has been reduced to 16 Mhz to allow use of a 16 Mhz in-circuit emulator (ICE). The microprocessor includes a three stage pipeline to improve performance. The board contains three types of memory: dynamic RAM (DRAM), static RAM (SRAM) and EPROM. Since the clock speed has been reduced, the memory access times listed in the SBC386/31 board hardware reference manual do not apply for this modified configuration. SRAM is used in the on-board 64K cache and is transparent to the user – no configuration options are available to alter it. The DRAM is configurable and occupies the lower 1 Mb of address space. The memory access times of the DRAM can vary as follows:

- 312.5 nsec (3 wait-states) for DRAM access with no pipelining
- 250 nsec (2 wait-states) for DRAM access with pipelining
- 125 nsec (0 wait-states) for SRAM cache access

The EPROMs used are Intel type 27512 and provide 128 Kb of ROM at the board's highest addresses (FFFE0000 to FFFFFFFF). The board is configured to have 6 wait-states giving an access time of 500 nanoseconds with the board running at 16 Mhz.

HP LOGIC ANALYZER AND 80386 SOFTWARE

All timing measurements were accomplished using a Hewlett-Packard 1650A logic analyzer with an 80386 microprocessor preprocessor interface and inverse assembly software. This configuration allows the capture of the interrupt acknowledge signal given by the microprocessor and the disassembly of every machine language instruction executed between the interrupt and the beginning of the interrupt handler. It also allows the capture of all the instructions executed between the end of the interrupt handler and the point in the main program where execution was interrupted and resumed.

The logic analyzer allows time-stamping of each instruction executed and has a resolution of 10 nanoseconds. Timing measurements are gathered by comparing the printed output of the logic analyzer to the actual code generated by the compiler to verify where the interrupt handler was and then subtracting the difference in the time-stamps. This method is used to measure the time both to enter and to exit the handler.

```
with SYSTEM; use SYSTEM;

package SHARED_DATA is

    INTERRUPT_SIGNAL : constant := 1;
    type COMMON_DATA_TYPE is record
        ELEMENT_1 : integer;
        ELEMENT_2 : integer;
    end record;
    COMMON_DATA : COMMON_DATA_TYPE;
    for COMMON_DATA use at address'ref (16#F0_000#);

end SHARED_DATA;

with SYSTEM; use SYSTEM;
with SHARED_DATA; use SHARED_DATA;
with INTR_INTERFACE;
with MACH_CONFIG;
with MACHINE_CODE;

package body HANDLER_EXAMPLE is

    HANDLER_ID : constant := 130;
    INTERRUPT_SIGNAL : constant := 1;
    INT_SIGNAL_ADDR : address;
    ATTACH_NO_ADDR : address := NO_ADDR;
    DETACH_NO_ADDR : address := NO_ADDR;

    procedure PROCESS_INTERRUPT is
    begin
        -- code for interrupt handler is inserted here
        -- modify common_data here
        intr_interface.POST_SIGNAL(INT_SIGNAL_ADDR);
    end PROCESS_INTERRUPT;

    procedure INTERRUPT_HANDLER is
        pragma IMPLICIT_CODE (off);
        use MACHINE_CODE;
    begin
        -- Push all remaining registers (SS, ESP, EFLAGS,
        -- CS, and EIP are already saved by interrupt)
        code_1'(push, DS);
        code_1'(push, ES);
        code_0'(op = pushad);
        -- Switch to kernel data segment (CS and SS are
        -- already switched to kernel segments by interrupt)
        code_2'(mov, AX, +mach_config.KRN_DATA_SELECTOR);
        code_2'(mov, DS, AX);
        code_2'(mov, ES, AX);
        -- Process interrupt
        code_1'(call, PROCESS_INTERRUPT'ref);
        -- Complete interrupt
        code_1'(call, intr_interface.COMPLETE_INTERRUPT'ref);
        -- No return back here
    end INTERRUPT_HANDLER;

begin

    intr_interface.REPLACE_VECTOR(HANDLER_ID, INTERRUPT_HANDLER'address);
    INT_SIGNAL_ADDR := intr_interface.CREATE_SIGNAL
        (INTERRUPT_SIGNAL, ATTACH_NO_ADDR, DETACH_NO_ADDR);

end HANDLER_EXAMPLE;
```

Figure 2. Sample Verdix Interface.

```
with SHARED_DATA; use SHARED_DATA;
with USER_STATUS;

package body EXAMPLE_PACKAGE is

    task EXAMPLE_TASK is
        entry TASK_SIGNAL;
        for TASK_SIGNAL use at address'ref(INTERRUPT_SIGNAL);
    end EXAMPLE_TASK;

    task body EXAMPLE_TASK is
        OLD_STATUS : USER_STATUS.STATUS_T;
    begin
        loop
            accept TASK_SIGNAL;
            USER_STATUS.DISABLE_INTERRUPTS(OLD_STATUS);
            -- code for interrupt handler
            -- read or modify common_data
            USER_STATUS.ENABLE_INTERRUPTS(OLD_STATUS);
        end loop;
    end EXAMPLE_TASK;

end EXAMPLE_PACKAGE;
```

Figure 2. Sample Verdix Interface (cont'd).

SUMMARY OF COMPILERS TESTED

DDC-I V4.2 (80386 TARGET)

TI has been using a protected mode version of the compiler, but DDC-I also sells a version that operates in the real addressing mode of the 80386. The Ada compiler and linker are shipped with several extra tools, including a Program Library Utility, an EDT Editor Interface (EDA), a disassembler, and an extract tool. DDC-I also offers an optional cross debugger with windowing hosted on a VAX and an optional DARTS runtime system with Hard Deadline scheduling capability.

The DDC-I compiler does require the use of VAX hosted Intel or compatible 80386 development tools, such as ASM386, BLD386, BND386, LIB386 and MAP386, which must be purchased separately. An alternative to the cross debugger for downloading is an In-Circuit Emulator, which is the method TI uses for debugging and executing. All source listings are provided by DDC-I so that the compiler can be configured for any 80386 target hardware.

INTEL V3.20.03 (80386 TARGET)

This compiler is a full production release and has been validated. Intel is providing customer support for the compiler although it was originally developed by Telesoft. The major components of the Ada-386 development system include an Ada compiler, library manager, binder, linker, object module tools, global optimizer and source level debugger. The system also includes numerous language tools including a cross reference utility, Ada source dependency lister, and source formatter (pretty printer). The execution environment is tailorable by the user for execution on several 80386 target machines.

VERDIX V5.5 (80386 TARGET)

The Verdix Ada Development System (VADS) for Intel 80386 targets is a full production release, validated Ada compiler. The system includes the software components for library management, compilation, program generation, object file analysis, debugging, source code formatting, and additional tools and libraries.

RESULTS

This section of the report presents interrupt response time measurements for the three compilers discussed previously.

Figures 3 through 5 show the measured interrupt entry and exit time for each interrupt service method supported by each compiler.

In Figure 3, the value shown for *fast interface 1* includes exit code to check the timer; *fast interface 2* shows the exit code with the timer check removed. See the section on *Analysis of Generated Code* for additional information.

Figure 4 shows the fast and slow response times for the Intel compiler. It appears that the unusually fast entry time for the interrupt handler is the result of the synchronization optimization mentioned in section 2.4.2. A longer interrupt entry time would be likely if the task had not been ready to accept the interrupt. Likewise, the exit time for the fast handler is just as long as the slow handler if the handler is not in a loop and rescheduling is done on exit.

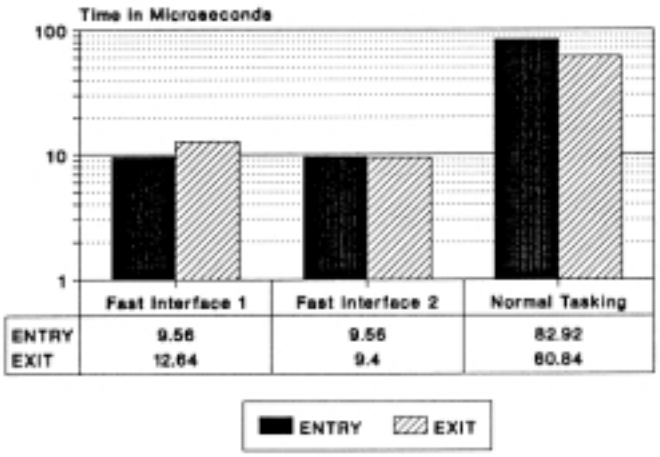

Figure 3. DDC-I.

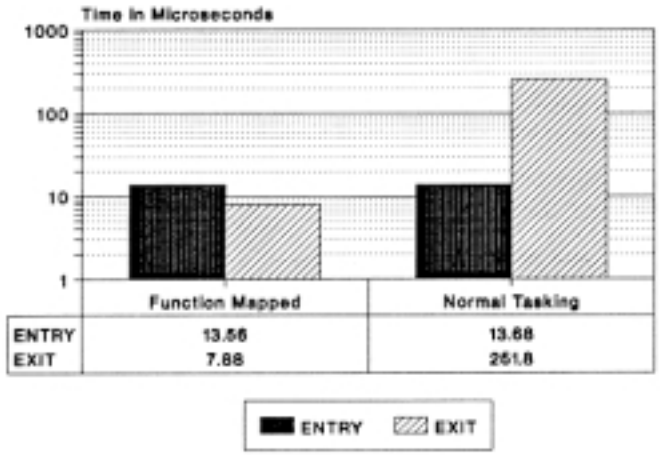

Figure 4. Intel.

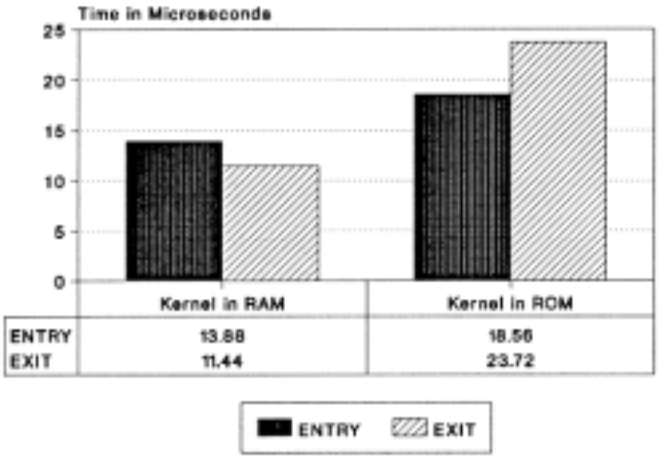

Figure 5. Verdix.

Figure 5 shows the results for Verdix on the 80386. The same interrupt handler was measured with the runtime kernel executing from EPROM and from DRAM. The interrupt response time for the signal mechanism was not measured.

Figure 6 shows the fastest combined interrupt entry/exit times for each of the 80386 targeted compilers. As of this writing, the DDC-I compiler shows the best response times of all compilers tested. Recall that the original objective of this report was to establish whether or not Ada is capable of handling embedded

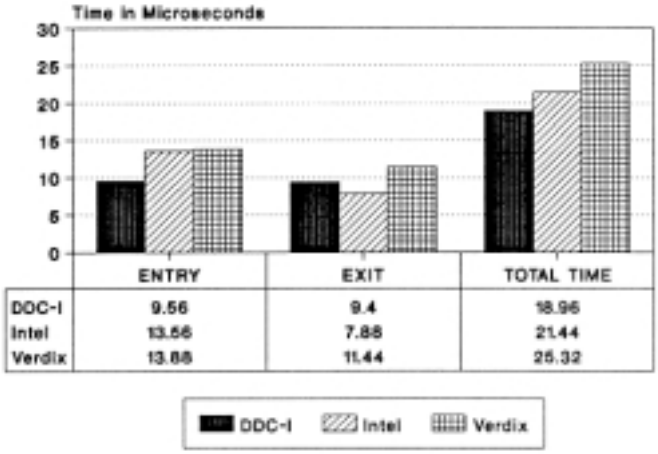

Figure 6. Fastest Combined Entry/Exit Times.

applications with high interrupt rates. The best combined entry/exit time measured for this report was 18.96 microseconds and was observed using the DDC-I Ada-386 compiler. It should be noted that reported times can be affected depending on whether or not the measured code segment is completely cache resident. For this report, memory caches were *enabled* on those targets that used them. None of the benchmarks were executed with caches *disabled*.

Figure 7 shows anticipated throughput values for an application using an interrupt handler whose duration ranges from 0.2 to 10 milliseconds. Each ordinate value in the graph is computed by first

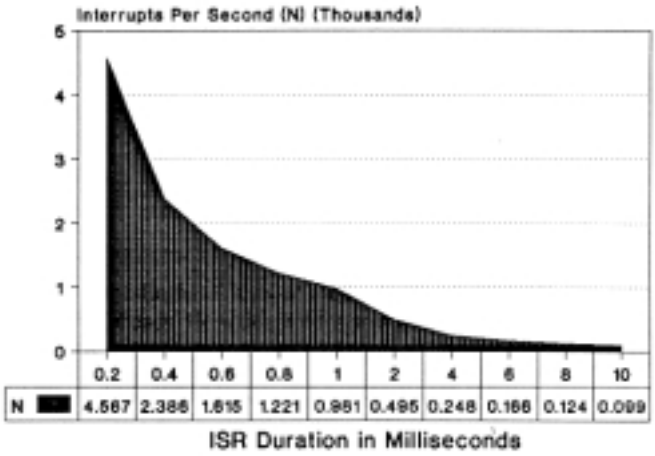

Figure 7. Interrupt Throughput.

adding 18.96 microseconds to the corresponding x-value (converted to microseconds) and then calculating the maximum number of interrupts per second. For example, to find the maximum number of interrupts per second that can be processed by an interrupt handler with a duration of 1 millisecond:

$$10^6 / (18.96 + (1 \times 10^3)) = 981$$

Put another way, if entry to an interrupt handler requires 9.56 microseconds, exit from the handler requires 9.4 microseconds,

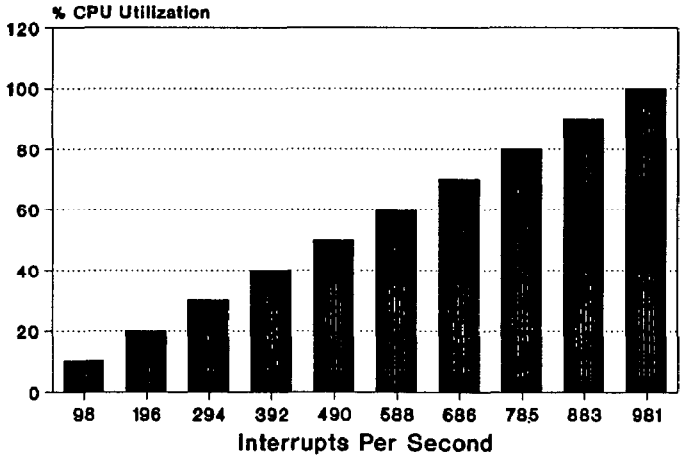

Figure 8. CPU Utilization.

and the interrupt service routine itself takes 1 millisecond, throughput will be approximately 981 interrupts per second.

Another way to view interrupt throughput is to consider how much of the CPU is being utilized with respect to interrupt-driven activity alone. Figure 8 shows percentage of CPU utilization as a function of the number of interrupts per second processed (for 1 millisecond ISR and 18.96 microsecond combined entry and exit time). For example, at 588 interrupts per second, CPU utilization is approximately 60%, leaving 40% of the CPU time available for other tasks.

ANALYSIS OF GENERATED CODE

The amount of overhead code generated for an interrupt handler varies between compilers and it is important to understand what actually happens in the context switch before and after servicing an interrupt.

Both Intel and DDC-I have a similar fast interrupt mechanism. Each makes an entry into the IDT that points to the code in the body of the accept statement for the interrupt handler. The code executed on entry consists of pushing registers on the stack and setting up the correct segments for code and data. Both compilers generate code to determine which interrupt is being handled. Intel checks for a fast interrupt before entering the handler, causing its entry time to be longer. DDC-I checks for the timer after the interrupt code is executed, but claims this is not always necessary – a machine code insertion can be made for only the code that is needed, avoiding the extra instructions. This results in a 3.2 microsecond improvement in exit time.

Verdix handles interrupts still differently. The registers are pushed on the stack, segments are set up, and a call is made to the interrupt handler. On exit, COMPLETE_INTERRUPT is called to handle popping of registers from the stack and resetting of the segment registers to correct values. The variance in the numbers shown in the comparison between the kernel being in RAM and in ROM reflect the proportion of the code that is executed in ROM, i.e., the more code in ROM, the longer it takes to execute (EPROM is accessed with six wait states while DRAM is accessed with at most three.)

For the compilers that support normal interrupt handling, the code generated includes all the overhead necessary for normal Ada tasking. The interrupt is handled by using a mechanism to associate the interrupt with an entry call to the task interrupt handler. This *simulated* entry call is pointed-to by the interrupt vector.

Additional details regarding the specific methods used by each compiler to implement its tasking functions were not investigated for this report.

RUNTIME TRENDS

Ada compilers and runtime systems continue to evolve, as do issues involving the applicability of Ada to real-time embedded systems. A longstanding concern among developers has been that Ada's existing mechanisms for handling multitasking, synchronization, interrupts, etc. are not efficient enough for real-time applications. In response to accusations of this type, vendors and special interest groups have worked toward defining custom runtimes and interfaces that allow more efficient use of Ada in time-critical applications.

BACKGROUND INFORMATION

The use of *non-Ada* interfaces raises a number of questions regarding portability, reuse, and reliability. In the strictest sense, vendors that provide customized runtime interfaces are encouraging the development of applications that are not 100% Ada. Furthermore, a customer may not accept applications that are not fully compliant with the existing Ada standard. Performance requirements imposed by the end user may not necessarily outweigh the importance of portability across multiple targets. Nevertheless, a case can be made for using custom runtime calls in those applications where meeting deadlines in real-time is critical. Calls to a custom runtime executive do present a more readable and less costly alternative to coding and integrating time-critical code sections in assembly language.

The remaining paragraphs in this section present a brief overview of various custom runtime systems developed by DDC-I, Intel, and Verdix.

VENDOR-SPECIFIC PROGRESS

DDC-I DARTS

The DDC-I Ada Runtime System (DARTS) for Intel 80x86 targets does not implement a custom procedural interface to its runtime environment. Instead, DDC-I's approach has been to highly optimize its runtime within the confines of the existing Ada tasking model. Based on the T-tests in the PIWG benchmark suite (see [1]), DDC-I's performance is two to four times better than either Verdix or Intel. Despite this already significant advantage in tasking efficiency, DDC-I provides extensions to its

runtime that support the notion of *hard deadline scheduling* – the ability to guarantee reliability of real-time applications using tasking while maximizing processor utilization. Marketing information [16-18] obtained from DDC-I summarizes the rationale for these extensions as follows:

"In a series of recent research papers, Carnegie-Mellon University (CMU) has recommended that rate monotonic scheduling be used by tasking applications to guarantee meeting critical processing deadlines. Rate monotonic scheduling is a simple algorithm which assigns higher priorities to more frequently executed tasks. ... CMU identified four issues in the Ada language which limit use of rate monotonic scheduling: fixed task priorities, arbitrary SELECT alternatives, FIFO queues, and priority inversion. ... DDC-I has implemented the four solutions recommended by CMU which permit full use of rate monotonic scheduling in Ada [by adding support for] dynamic task priorities, priority SELECT alternatives, priority queues [and] priority inheritance."

According to DDC-I, measurements of rate monotonic scheduling demonstrate that processor utilization can exceed 90% in some systems before critical deadlines are missed, where only 30% processor utilization was achieved without rate monotonic scheduling.

The hard deadline extensions are provided with a source code license to the DDC-I-Ada runtime system.

INTEL IRMK

iRMK is Intel's real-time kernel for the 80386. The iRMK interface is a separately orderable unit and consists of the *pre-IMPORTed* iRMK kernel plus all the needed interface specs and documentation. According to marketing information published by Intel, iRMK features deterministic memory allocation, bounded interrupt latency times, and fast, efficient task synchronization mechanisms. In addition, iRMK offers direct access to Multibus-II resources including message passing.

iRMK defines a *calls interface* to support real-time programming in Ada. In particular, the calls interface allows the application programmer to use either Ada tasking or the iRMK kernel tasking, but not both at the same time (this restriction to be removed in release 3.0).

The iRMK real-time kernel (version 1) provides custom interfaces for task, interrupt, and time management, as well as intertask communication and synchronization, Multibus II message passing and Multibus II interconnect space management.

Intel is in the process of negotiating a Release 3.0 product that will incorporate iRMK with the Ada-386 runtime environment. With this release, Ada calls to the Ada runtime environment will map into iRMK primitives (for example, the **delay** statement will invisibly map onto KN_SLEEP). Direct calls to iRMK will still be supported.

VERDIX VADS AND VRTX

The Verdix Ada Development System (VADS) currently supports Ready System's VRTX (not evaluated in this report) as well as their own proprietary executive. The latter contains the package MACHINE_CODE described in section 13.8 of the Ada LRM. Machine-code insertions provide, from within the Ada language, low-level access to processor features that are normally only accessible from assembly language. According to marketing information received from Verdix, VADS machine-code insertions provide features such as the 'REF attribute, a full range of addressing modes, and parameters that enable the programmer to integrate the machine-code into surrounding code with minimal effort.

SUMMARY AND CONCLUSIONS

Based on measurements of interrupt throughput shown earlier in this report, it appears that Ada compiler and runtime technology have reached a level of maturity that can support embedded applications with relatively high interrupt rates. In particular, ***for interrupt service routines ranging from 1.0 to 0.2 milliseconds duration, an Ada application can realize throughput ranging from 1000 to 4500+ interrupts per second using the Intel 80386.*** However, this level of throughput can only be demonstrated today using custom interrupt handler interfaces or runtimes that circumvent much of the overhead associated with the standard Ada tasking model. This implies non-portable, processor-dependent code that *may* not be acceptable to the end user.

Consideration has been given to measurements in a single interrupt scenario. There are still questions to be answered regarding quantitative measurements of multiple, nested interrupt scenarios such as those required in a GPS receiver application. To model this type of application in the lab requires the ability to stimulate the external interrupt lines in real-time. Although this approach was not used to obtain data for this particular report, the single interrupt measurements still have considerable value in helping to establish a baseline for anticipated performance.

REFERENCES

1. Texas Instruments Incorporated, Software Systems Technology. *Ada Compiler Benchmark Report Version 1.1.* October 1988.

2. N. Weiderman. *Ada Compiler Selection Handbook Version 1.0 (DRAFT).* Real-Time Embedded System Testbed Project, Software Engineering Institute, March 1989 (unpublished technical report).

3. D. Bryan. *"Dear Ada".* Ada Letters, July-August 1988, Volume VIII, Number 4.

4. J.R. Hunt. *Interrupts and Ada.* Ada Letters, 1988 Special Edition, Volume VIII, Number 7.

5. Ada Joint Program Office. *Ada Programming Language (ANSI/MIL-STD-1815A-1983).* Department of Defense, Ada Joint Program Office, Washington, D.C., February 17, 1983. The official reference manual for the Ada programming language.

6. E.W. Olsen and S.B. Whitehill. *Ada for Programmers.* Reston Publishing Company, Inc. Reston, Virginia; 1983.

7. Intel Corporation. *Ada-386 User's Guide for VAX/VMS Systems.* Order No. 481048-002; 1988.

8. Verdix Corporation. *VADS Verdix Ada Development System Version 5.5 for VMS iAPX80386,* VAda-010-03315; November 1987.

9. DDC-International. *DDC-I Ada Compiler System User's Guide for DACS-80x86,* Document No. DDC-I 5801/RPT/62, Issue 9, April 18, 1988.

10. Intel Corporation. *iSBC 386/31 Single Board Computer Hardware Reference Manual*, Order No. 453652-001, 1987.

11. D. Cornhill and L. Sha. *Priority Inversion in Ada or What Should be the Priority of an Ada Server Task?* Carnegie-Mellon University, Department of Computer Science. Internal research paper.

12. D. Cornhill, et al. *Limitations of Ada for Real-Time Scheduling.* Carnegie-Mellon University, Department of Computer Science. Internal research paper.

13. L. Sha, J. P. Lehoczky, R. Rajkumar. *Task Scheduling in Distributed Real-Time Systems.* Carnegie-Mellon University, Department of Computer Science. Internal research paper.

14. Texas Instruments Incorporated, Software Systems Technology. *A Real-Time Programmers Guide to Ada,* SP43-EG87, December 1987.

ABOUT THE AUTHORS:

DAVID G. STRUBLE is the Ada Technology Section Manager in the Military Computer Systems Department at Texas Instruments and is responsible for steering Ada support activities within TI. He received a B.S. degree in mathematics and computer science from the University of Dayton and the M.S. degree in computer and information sciences from The Ohio State University. Author's Present Address: Texas Instruments Incorporated, P.O. 869305 Mail Station 8435, Plano, TX 75086.

MICHAEL J. WAGNER is a member of the Ada Applications Work Group, Ada Technology Section, in the Military Computer Systems Department at Texas Instruments. Mr. Wagner is responsible for evaluation of Ada compilers and target architectures for use in embedded weapon systems. He received a B.S. degree in electrical engineering from Iowa State University in 1988. Author's Present Address: Texas Instruments Incorporated, P.O. Box 869305 Mail Station 8435, Plano, TX 75086.