Edgar H. Sibley
Panel Editor

Benchmarks are developed to measure the Ada notion of time, the Ada features believed important to real-time performance, and other time-related features that are not part of the language, but are part of the run-time system; these benchmarks are then applied to the language and run-time system, and the results evaluated.

TOWARD REAL-TIME PERFORMANCE BENCHMARKS FOR ADA

RUSSELL M. CLAPP, LOUIS DUCHESNEAU, RICHARD A. VOLZ, TREVOR N. MUDGE, and TIMOTHY SCHULTZE

As stated in the forward to the Ada Language Reference Manual [1], "Ada is the result of a collective effort to design a common language for programming large scale and real-time systems." The common denominator among real-time systems (e.g., the avionic system in an airplane, a robot controller, and even the controller for a video game) is the need to meet a variety of real-time constraints. Although Ada® is intended to be used for such real-time applications, there is nothing in the Language Reference Manual (LRM) that ensures that Ada programs, regardless of processor speed, will provide the performance necessary to accommodate the real-time constraints of particular applications. The Ada Compiler Validation Capability (ACVC) suite of programs was established to validate the form and meaning of programs written in Ada, but not to specify the size or speed of their object code, or the precise nature of their task scheduling mechanisms, all of which are critical to real-time performance. In other words, the Ada language contains mechanisms to accommodate real-time applications, but leaves performance issues to supplemental measurement. This article addresses the issue of real-time performance measurement—particularly time measurement and scheduling for which adequate requirements are *not* explicitly stated in the LRM—through the design and use of a set of benchmarks that measure the real-time performance of code produced by an Ada compiler.

Benchmarking can be approached in two ways: by developing a *composite* benchmark (e.g., Whetstone or Dhrystone [5, 9]); or developing a *set* of benchmarks, each of which measures the performance of a specific feature of the implementation [6]. Although the composite benchmark is easier to apply, no single composite can capture all of the information required for even a modest spectrum of real-time applications. Moreover, since detailed knowledge of the performance of individual features is often required for applications planning, and is also useful in understanding the relation between real-time performance, language constructs, and compiler implementation, our approach concentrates on techniques for measuring the performance of individual language features.

Measuring the performance of individual language features through benchmarks involves a number of complex operations, including

- isolating the feature to be measured;
- achieving measurement accuracy and repeatability;
- eliminating underlying operating-system interference from time slicing, daemons, and paging;

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

This work was partially sponsored by Land System Division of General Dynamics and NASA.

each of which is considered in this article. Besides the performance of individual language features, there are other real-time performance measurements associated with the run-time system (e.g., measurements of scheduling and storage management algorithms).

In this article, we focus on the features from the language and run-time system that are believed to be important for real-time performance, concentrating not only on the benchmarks, but also on the basic measurement techniques used. A comprehensive effort to acquire benchmark programs and provide an extensive database of comparative results on all major Ada compilers is being conducted under the auspices of the ACM Special Interest Group in Ada [8]. Most of the benchmark tests presented here were contributed to that effort in the summer of 1985; the remainder (i.e., those developed during the fall of 1985) were contributed in early 1986.

The development and interpretation of measurement techniques for real-time programming are based on the Ada notion of time, which is reviewed in the following section. Thereafter, techniques are presented for achieving basic measurement accuracy, isolating the features to be measured, and determining the interference of operating-system functions. A subsequent section presents the set of features believed important for real-time performance, discusses why they are considered important, and describes the measurements to be made by the benchmark. Particular focus is given to scheduling operations and time measurements. The next section presents the results of the benchmark tests for several compilers: Verdix Versions 4.06, 5.1, and 5.2 running with Unix® 4.2 bsd on a VAX 11/780; DEC VAX Ada Version 1.1 running with Micro VMS 4.1 on a Microvax II; DEC VAX Ada Version 1.3 running with VMS 4.4 on a VAX 11/780; and Alsys Version 1.0 running with Aegis Version 9.2 on an Apollo DN660. It is important to note that these versions of the compilers are intended for time-shared use, *not* real-time applications, and therefore the results should not be interpreted with real-time performance in mind. At the time of this writing, however, these were the principal Ada compilers available to the authors, and the results do help illustrate the methods presented. The parameters obtained also indicate areas in which users should look for improvements in cross-compilers intended for real-time applications.

REVIEW OF ADA TIME UNITS

The Ada LRM defines several entities that relate to time, its representation within Ada programs, and the execution of Ada programs:

- the data type TIME, objects of which are used to hold an internal representation of an absolute point in time;
- the data type DURATION, objects of which are used to hold values for intervals of time;
- a predefined package, CALENDAR, which provides functions to perform arithmetic on objects of type TIME or DURATION;
- a predefined function, CLOCK, which returns a value of type TIME corresponding to the current time;
- DURATION'SMALL, which gives an indication of the smallest interval of time that can be represented in a program; this time interval must be less than or equal to 20 ms, with a recommendation that it be as small as 50 μs;
- the value SYSTEM.TICK, which is defined as the basic system unit of time;
- the operation **delay**, which allows a task to suspend itself for a period of time.

The semantics associated with the first three of these entities are clear, whereas those of the last four warrant some discussion.

Specifically, values of type DURATION are fixed-point numbers and thus are integer multiples of the constant DURATION'SMALL. DURATION objects are only *data representations* of time and in no way imply actual performance of a system for time measurements or scheduling. That is, there is no required relation between the clock resolution time and DURATION'SMALL. For example, on the Verdix and Telesoft compilers for a VAX Unix system, DURATION'SMALL is 61 μs, while the timer resolutions are 10 ms and 1 s, respectively.

The CLOCK function generally presumes an underlying clock or timer that is periodically updated at some rate undefined by the LRM. We call this period the *resolution* time of the system. CLOCK simply returns the value of time associated with the current value of the underlying timer. If the execution time of CLOCK is less than the time resolution, successive evaluations of CLOCK may return the same value.

The term "basic system unit of time" is not very specific. One might think it refers to the basic CPU clock cycle. However, the constant SYSTEM.TICK is used by several compiler vendors to hold the value of the resolution of time measurements available from the CLOCK function.

In addition to the above, an implementation may have other important time-related parameters that are not identified in the LRM. For example, some validated Ada implementations frequently insert sizable delays—in conjunction with the **delay** statement—that are neither directly specified by the pro-

Unix is a trademark of AT&T Bell Laboratories.

grammer nor caused by system load, but are present simply for convenience in the implementation of the compiler and run-time system. Parameters that fall in this category will be identified in the discussion that follows, and techniques will be proposed for measuring them.

MEASUREMENT TECHNIQUES

There are two basic techniques for measuring the time needed to perform an operation. The first is to isolate the operation and make time measurements before and after performing it; however, for this to be adequate, the time resolution of an individual measurement must be considerably less than the time required by the operation being measured. Unfortunately, this is typically not the case, and an alternative method must be found. The second technique, and the one used here, involves executing the operation a large number of times, taking time readings only at the beginning and the end, and obtaining the desired time by averaging.

Although this sounds simple and straightforward, there are a number of complications that must be handled carefully if the results obtained are to be meaningful:

- isolating the feature to be measured and avoiding compiler optimizations that would invalidate the measurement,
- obtaining sufficient accuracy in the measurement,
- avoiding operating-system distortions, and
- obtaining repeatable results.

Isolation of Features

To isolate a specific feature from other features of the language for measurement purposes, the basic technique is to use two execution frames—a control loop and a test loop—which differ only by the feature whose execution time is being measured. Theoretically, a difference of execution times between the control loop and the test loop yields the time of the function being measured. Code optimization, however, can distort benchmark results by removing code from test loops, eliminating procedure calls, or performing folding. The benchmark programs must therefore utilize techniques to thwart code optimizers.

The key to avoiding these problems is not to let the compiler see constants or expressions in the loops whose times are being measured. For example, instead of using a **for loop** with a constant iteration limit, a **while loop** is used with the termination condition being the equality of the index variable to an iteration variable. The index variable is incremented by a procedure, the body of which is defined in the body of a separate package. The iteration variables are declared and initialized in the specification of a library package. Since the iteration values are kept in variables (not constants) and the body of the increment procedure is hidden in the body of the package, there is no way the benchmark loops can be removed by optimization as long as the package specification and body are compiled separately, with the body being compiled after the benchmarking unit.

Similarly, the compiler must be prevented from either removing the execution of the feature being tested from the loop, or eliminating the loop entirely from the control loop that does not contain the feature. To ensure that these problems do not arise, control functions are inserted into both loops, and the feature being measured is placed in a subprogram called from a library unit [3]. Again, if the bodies of these subprograms are compiled separately, and after the benchmark itself, the compiler is unable to determine enough information to perform optimization and remove anything from either the control or test loops. These techniques will be evident in the benchmarks described below.

The loops must each be executed N times, as discussed in the next section, to produce the desired accuracy. The form of the test loop is

```
T1 := CLOCK;
while I < N loop
   control functions;
   DO_SEPARATE_PROC_F ; -- the function F
        -- whose time is being measured
   INCREMENT( I );                                  (L1)
end loop;
T2 := CLOCK;
T^m := T2 − T1;
```

The control functions and subprogram call to increment I are included to thwart code optimizers. The control frame would be identical to this except that a separately compiled function DO_SEPARATE_PROC_NULL would replace DO_SEPARATE_PROC_F.

Basic Measurement Accuracy

Knowledge of both the resolution of a time measurement and the variability of the time needed to make a time measurement is required to determine the number of iterations needed to obtain a parameter measurement within a given tolerance. If τ is the basic time resolution unit in terms of which all time measurements are made, then the value returned by the CLOCK function at time t is

$$\left\lfloor \frac{t + \tau_c \pm \tau_v}{\tau} \right\rfloor \cdot \tau \quad , \tag{1}$$

where $\lfloor x \rfloor$ is the "floor" function (the largest integer

less than or equal to x), τ_c is the nominal time required to perform the CLOCK function, and τ_v is a variable indicating a (hopefully) small random variation in the time required to perform the CLOCK function. Since a difference of CLOCK measurements will be used, τ_c will subtract out of the equations to be developed and can be ignored. In all of the equations that follow, it is assumed that τ_v is small in comparison to τ and can also be ignored. In any application, however, this assumption must be verified. One of the tests described under "Features to Be Measured" on page 768 can be used for this verification.

If the time required to execute the loop excluding F is T_0 and the time required to perform function F is T_F (i.e., T_F is the time we are trying to ascertain), then the difference between the values returned by the two calls to the CLOCK function above will be

$$T^m = N(T_0 + T_F) \pm \delta \cdot \tau \quad , \tag{2}$$

where

$$0 \le \delta < 1 \quad .$$

Then T_F is given by

$$T_F = \frac{T^m}{N} - T_0 \pm \frac{\delta \cdot \tau}{N} \quad . \tag{3}$$

Thus, the accuracy of the measurement is determined by

$$\frac{\delta \cdot \tau}{N} < \frac{\tau}{N} \quad . \tag{4}$$

Once the time resolution unit τ is determined, the number of iterations can be chosen to provide the desired accuracy. However, the impact of cumulative error buildup must be taken into account; if T_0 is obtained by a similar type of measurement, one must increase N for both measurements.

To measure τ, a call to the CLOCK function is placed in a loop that is executed a large number of times. Each time value obtained is placed in an array. We will now show that the second difference of the values obtained will equal either zero or the time resolution unit.

Let the time to complete one execution of the loop be

$$T_{loop}(1) = n \cdot \tau + \delta \cdot \tau \quad \text{where} \quad n \text{ is an integer and } 0 \le \delta < 1 \quad . \tag{5}$$

Without loss of generality, consider that the first execution of the loop begins at time zero. Then the time at the end of the kth iteration will be

$$T_{loop}(k) = k \cdot n \cdot \tau + k \cdot \delta \cdot \tau \tag{6}$$

and the measured time will be

$$T^m(k) = k \cdot n \cdot \tau + \lfloor k \cdot \delta \rfloor \cdot \tau \tag{7}$$

since the times returned are a multiple of the CLOCK resolution τ. The first difference of the measured times can be written

$$\Delta T^m(k) = T^m(k+1) - T^m(k) \tag{8}$$
$$= n \cdot \tau + \{\lfloor (k+1) \cdot \delta \rfloor - \lfloor k \cdot \delta \rfloor\} \cdot \tau \quad .$$

We note that since k is an integer and δ lies in [0, 1) we have

$$\lfloor (k+1) \cdot \delta \rfloor - \lfloor k \cdot \delta \rfloor = 0 \text{ or } 1 \quad . \tag{9}$$

Therefore, in the second difference of the times measured by the CLOCK function, the $n \cdot \tau$ in Equation 8 will subtract out and yield

$$\Delta T^m(k) - \Delta T^m(k-1) = 0, \tau \text{ or } -\tau \quad . \tag{10}$$

More specifically, the second difference will yield one of the following sequences:

$$\ldots, 0, \tau, -\tau, 0, \ldots, 0, \tau, -\tau, 0, \ldots \tag{11a}$$

or

$$\ldots, 0, -\tau, \tau, 0, \ldots, 0, -\tau, \tau, 0, \ldots \tag{11b}$$

depending on the value of δ. The length of the substrings of zeros is approximately

$$L_0 = \begin{cases} \dfrac{1}{\delta} - 2 & \text{if } 0 < \delta \le \dfrac{1}{2} \\ \dfrac{1}{1-\delta} - 2 & \text{if } \dfrac{1}{2} < \delta < 1 \end{cases} \quad . \tag{12}$$

L_0 can be controlled by empirically adding instructions to the loop. If $\delta < 1/2$, the sequence (11a) results, whereas, if $\delta > 1/2$, (11b) is obtained.

If n in the above equations is zero, then a first difference measurement will suffice, yielding a string of zeros with τ appearing occasionally. The only purpose in taking the second difference was to eliminate n.

This second differencing procedure represents a reliable technique for measuring the resolution time of the CLOCK function. As will be seen below, it is also useful for measuring a number of other parameters associated with real-time system performance.

Operating-System Interference

Isolating the feature being measured from other language features and from code optimization is not the only isolation that must be achieved. The *timing* of the feature being measured must also be isolated from the times for other user processes or for the operating system itself. Since the CLOCK function measures absolute time, any other processes executing during the test (e.g., in a time-shared mode) would contribute to the measured time and thus

distort the results. Some operating systems (e.g., Unix) provide a timing function that nominally measures only the time of the processes being tested, excluding the times of the operating systems or other user processes. Not all operating systems can be expected to have this function, however, and even for those that do, there is some question as to how precisely this calculation is made. Benchmark tests should therefore be run on a system with no other user processes in concurrent execution and with all daemon processes disabled. A consequence of this requirement is that no output should be generated by a benchmark until all timing is completed, since a request for output could create an independent process running concurrently with the benchmark.

Even with the disabling of daemon processes and running on a single user system, there are still timing anomalies that must be detected and measured, most notably the time-sharing activities of the operating system. The operating system can still be expected to interrupt the benchmark periodically to check the queue for other processes waiting to run, and then return control to the benchmark process. For sufficiently high use of memory, operating-system paging functions may also be invoked. However, except for memory allocation/deallocation tests, benchmarks can usually be designed to use less memory than the size that will cause paging activity. The frequency and duration of these operating-system actions must be determined and taken into account in the timing calculations.

We begin by analyzing the effect of a function F_{os}, which periodically intrudes on the operation of the benchmark. Let the function F_{os} require a constant T_{os} seconds and occur with period T_p, and make the following definitions:

T_c = actual time required to execute the control loop, N times;
T_{cf} = actual time required to execute the control loop and F, N times;
n_c = number of times F_{os} is executed during T_c;
n_{cf} = number of times F_{os} is executed during T_{cf};
T_c^m = measured time for T_c;
T_{cf}^m = measured time for T_{cf}.

It then follows that

$$T_c = N \cdot T_0 + n_c \cdot T_{os} \tag{13}$$

$$T_{cf} = N \cdot (T_0 + T_F) + n_{cf} \cdot T_{os} \quad . \tag{14}$$

Since the measured times must be multiples of the time resolution τ, we have

$$T_c^m = T_c + \delta_c \cdot \tau \tag{15}$$

$$T_{cf}^m = T_{cf} + \delta_{cf} \cdot \tau \tag{16}$$

where $-1 < \delta_c, \delta_{cf} < 1$. Then, letting the calculated time difference be $T_d = T_{cf}^m - T_c^m$, it is straightforward to obtain

$$T_F = \frac{T_d}{N} - \frac{(n_{cf} - n_c)}{N} \cdot T_{os} - (\delta_{cf} - \delta_c) \cdot \frac{\tau}{N} \quad . \tag{17}$$

Next, we observe that n_c and n_{cf} must be integers and hence that

$$n_c = \frac{T_c}{T_p} + \epsilon_c \tag{18}$$

$$n_{cf} = \frac{T_{cf}}{T_p} + \epsilon_{cf} \tag{19}$$

for some $-1 < \epsilon_c, \epsilon_{cf} < 1$. It can then be found that

$$T_F = \frac{T_d}{N} \cdot (1 - \beta) + 2 \cdot \epsilon \cdot \frac{T_{os}}{N} + 2 \cdot \delta \cdot (1 - \beta) \cdot \frac{\tau}{N} \tag{20}$$

for some $-1 < \delta, \epsilon < 1$ where

$$\beta = \frac{T_{os}}{T_p} < 1 \quad .$$

The two right-hand terms in Equation 20 can be made arbitrarily small by making N sufficiently large. The effect of β shows that the results previously obtained in Equation 3 are pessimistic and that a correction can be applied if T_p and T_{os} can be determined.

Estimates of T_p and T_{os} can be obtained by the same second differencing technique described above for obtaining the resolution time of the CLOCK function. Assume, for the moment, that T_{os} satisfies the relation $T_{os} \gg \tau$, that $T_p = m \cdot \tau$ for $m \gg 1$, and that δ in Equation 5 is zero. The latter assumption means that the contribution to the second difference from the resolution time τ is also zero, and the following analysis will reflect only the effects of T_{os}. From a filtering point of view, the time measurements are simply a staircase input to the simple second difference filter. The output string, then, is just

$$\ldots, 0, T_{os}, -T_{os}, 0, \ldots, 0, T_{os}, -T_{os}, 0, \ldots \quad . \tag{21}$$

This yields T_{os} directly, and the periodicity of the sequence gives the frequency of the operation T_p.

If $\delta \neq 0$, the above sequence will have the sequence of Equation 11 superimposed upon it, which may occasionally distort the value of T_{os} by $\pm \tau$. Further, if T_{os} is not an integral multiple of τ, the values in the sequence will only be within τ of T_{os}. If $T_{os} \gg \tau$, reasonable estimates of the parameters should still be obtainable. Theoretically, it is possible to derive the precise value of T_{os} based on the number of periods in Equation 21 that lie between

fluctuations of size τ in the nonzero values; in practice, however, it will be difficult to detect because of the length of sequence required and the distortion from the $(\tau, -\tau)$ occurrence as in Equation 11.

If $T_{os} < \tau$, it is again theoretically possible to obtain the measurements, but somewhat more difficult in practice. In this case, we begin by examining the sequence of Equation 11 and determining the length of the string of zeros between every $(\tau, -\tau)$ pair. If $T_c = 0$, then this length may not vary by more than 1. Any deviation by more than 1 indicates an occurrence of F_{os}. If $T_{os} < \tau$, this will be reflected by a shortening of the length of the string of zeros. The amount by which the string is shortened in a measure of T_{os} (measured in multiples of the loop time), and the period within which this is repeated indicates T_p.

Minor extensions of this technique permit the detection and evaluation of multiple periodic operating-system functions of differing service times. However, it is generally difficult to fit the execution time and period of more than a single function to the sequence of Equation 11. Nonetheless, by accumulating the shortening of the strings of zeros and dividing by total time, it is typically possible to get an overall estimate of the operating-system overhead involved.

Actual tests conducted using this approach revealed another difficulty. Some implementations of the CLOCK function involved the dynamic allocation of records, which in turn involved the invocation of a run-time system function. As discussed under "Dynamic Allocation of Objects" on page 766, the time required to perform this operation can vary widely, and this variation in storage allocation time will give the appearance of operating-system overhead. To avoid these problems, the Ada CLOCK function should not be used in tests to determine operating-system overhead. Instead, an implementation-dependent subprogram should be used that can read the system timer without invoking any variable time system functions such as storage allocation. Such a system-dependent subprogram was written and used in our tests. However, for all the other tests to be described, the CLOCK function is evaluated only at the beginning and end of a loop iterated a large number of times, and the effect of dynamic storage allocation is effectively eliminated, as shown in Equation 20. Thus, except for determining the operating-system overhead, the Ada CLOCK function may safely be used.

Resolution of Measurements

The result of Equation 20 is based on a periodically occurring function that always takes the same time to execute. Since in practice this assumption may not be entirely correct, repeated executions of the benchmark can be used to both test the validity of the assumptions and improve the accuracy of the results obtained.

The distribution of the estimates can be observed by running a repeated set of trials and then averaging the results obtained from each trial. The variance of the resultant estimate is divided by N_b if N_b trials of the benchmark are made.

An alternate strategy is to use the minimum of the values obtained. However, when this is done, it is important to determine the minimum of T_{cf} and T_c separately and use these values in the computation of T_d. Otherwise, one is likely to use a larger than average value of T_c in combination with a smaller than average value of T_{cf} and produce a result that is distorted on the side of being too small.

FEATURES TO BE MEASURED

In this section, we examine the features that are relevant to real-time execution and whose performance should be measured. A motivation is given for each proposed test as well as a precise statement of what is being measured. Where the measurement requires techniques beyond those just described, specific details are given.

The specific features discussed are listed below; all but the last three are measurements of features specified in the Ada LRM:

- subprogram calls;
- object allocation;
- exceptions;
- task elaboration, activation, and termination;
- task synchronization;
- CLOCK evaluation;
- TIME and DURATION evaluations;
- DELAY function and scheduling;
- object deallocation and garbage collection;
- interrupt response time.

In the areas of tasking, timing, and storage management, the compiler implementors have been given a great deal of implementation latitude, and as a result, it is difficult to develop a priori a set of benchmarks that completely characterize these areas. Since knowledge of the type of disciplines implemented is essential before a determination can be made as to what parameters it is relevant to measure, measurement techniques in these areas are oriented toward determining the general nature of the implementation techniques used.

Subprogram Overhead

With today's software running into sizes that exceed one million lines, modular programming is a necessity—but a necessity that also leads to an increase

in procedure and function calls. In a recent study, Zeigler and Weicker found that 26.8 percent of a typical Ada program, as implemented in the iMAX 432 system, consisted of subprogram calls [10]; Shimasaki et al. obtained a range of 26.5 percent to 41.4 percent for typical Pascal systems [7]. Since it is clear that the overhead associated with a subprogram call and return should not deter software producers from using a structured programming style, a way of avoiding this increased overhead is having the compiler generate an in-line expansion of the code of the subprogram where the call to it occurs. There is a trade-off here, however, in the sense that, as the call/return overhead is eliminated, the size of the object module is increased. Ada provides for a method of in-line expansion with the INLINE **pragma**, but a compiler is not required to implement this or any other **pragma**. By measuring both subprogram overhead and the time needed (if any) to execute code generated by an in-line expansion, one can determine whether or not the language/computer will encourage real-time systems programmers to use good programming techniques.

Several tests were designed to provide insight into the different aspects of subprogram calls. The first measures the raw overhead involved in entering and exiting a subprogram with no parameters and then determines the overhead associated with simple parameter passing by passing various numbers of INTEGER and ENUMERATION parameters. Composite objects may be passed either by copy or reference. Another test will determine which method is used, because, if the parameters are passed by reference, the time required will be independent of the number of components of the object. The final case involving parameters is the one in which the formal parameters of the subprogram are of an unconstrained composite type. The test in this situation is designed to measure the additional overhead present in passing constraint information along with the parameter itself. All of the tests include passing the parameters in the modes **in**, **out**, and **in out**.

All the tests involve two different types of subprogram calls, one to a subprogram that is a part of the same package as the caller, and the other to a subprogram in a package other than the one in which the caller resides. These two sets of tests determine if there is any difference in overhead between intra- and interpackage calls. In the case of intrapackage calls, the tests are repeated with the addition of the INLINE **pragma** to determine if the INLINE **pragma** is supported and, if it is, the amount of overhead involved in executing code generated by in-line expansion as opposed to executing the same set of statements originally coded without a subprogram call.

The final aspect of the tests involves the use of package instantiations of generic code. All the tests for interpackage and intrapackage calls are repeated with subprograms as part of a generic unit to determine the additional overhead involved in executing generic instantiations of the code.

Dynamic Allocation of Objects

Writing software without distinct bounds on the size of arrays and records or the number of tasks or variables improves portability and ease of support as the application changes. The ability to dynamically allocate objects is also important to the development of some algorithms, but with embedded real-time systems, the time required to dynamically allocate storage may make it an undesirable feature. To determine if dynamic allocation of objects is feasible in a real-time application, the associated overhead must be measured.

Three types of dynamic storage allocation are considered:

- allocating a fixed amount of storage by entering a subprogram or declare block with objects declared locally, where the amount of storage needed is known at compile time, but is allocated at run time;
- allocating a variable amount of storage not known at compile time (e.g., to an object like an array with variable bounds) by entering either a subprogram or declare block;
- achieving dynamic allocation explicitly with the **new** allocator, which can be used to allocate a single object of a particular type.

The overhead associated with each of these types of dynamic allocation is measured as follows.

In the case of fixed-length allocation, the times to allocate various numbers of objects of types INTEGER and ENUMERATION are measured as well as the times to allocate various sizes of arrays, records, and STRINGs. The objective is to determine the allocation overhead involved and whether the overhead differs based on the type of object allocated. With variable-length storage, arrays of various dimensions—bounded by variables—are allocated. The test in this case is designed to determine if allocation time depends on the size of the object. Many compilers will probably allocate small objects from the stack assigned to the task, and larger objects from the heap (which will typically take a much longer time). Finally, in the case of the **new** allocator, allocation time of objects of type INTEGER and ENUMERATION as well as composite type objects of various sizes is measured. This test will show if allocation time is dependent on size (in the composite type object case) and provide some idea as to the

relative efficiency of this method as opposed to the fixed-length case.

Exceptions

Embedded real-time systems require extensive error handling and recovery so that errors can be isolated and reported without bringing the whole system down. Also, modular programming encourages the abstraction of abnormal error reporting. Since many real-time systems must function in the absence of human intervention, as in spaceships, satellites, etc., the ability to provide extensive exception handling is of great importance.

The following four types of exception-handling routines—NUMERIC_ERROR, CONSTRAINT_ERROR, TASKING_ERROR, and user-defined exceptions—are interesting because they represent different ways in which exceptions are raised. The NUMERIC_ERROR exception is first discovered by the hardware and then propagated back to the run-time system by an interrupt signal from the hardware. The CONSTRAINT_ERROR is raised by the Ada run-time system. The TASKING_ERROR is raised during task elaboration, task activation, or certain conditions of conditional entry calls, and the user-defined exception is raised by the programmer. Except for the user-defined exception, the exceptions are raised either by forcing the relevant abnormal state in the code or by using the **raise** statement.

To gauge exception-handling efficiency, time measurements for responding to and propagating exceptions must be examined. The response time for an exception is the time taken between the raising of the exception and the start of exception-handler execution. When an exception is raised in a unit and no handler is present, the exception is propagated by raising the exception at the point where the unit was invoked. The time between raising an exception in a unit and its subsequent raising at the point where the unit is invoked is the time necessary to propagate the exception. In the tests presented here, both of these times are determined for three of the four types of exceptions mentioned above; the exceptions are raised both by the **raise** statement and by forcing the abnormal state to occur in the code.

Task Elaboration, Activation, and Termination

The tasking function represents the heart of the real-time power and usefulness of Ada. Many algorithms (e.g., buffering algorithms) involve the creation and execution of tasks such as the reader-writer scheme described in Barnes [2]. Nevertheless, task elaboration, activation, and termination are almost always suspect operations in real-time programming, and programmers often allocate tasks statically to reduce run-time execution time. As a result, exploring the efficiencies of task elaboration and activation is of special interest.

In this test, the time measured is that consumed elaborating a task's specification, activating the task, and terminating the task. This composite value gives an indication of the overhead involved in the use of the tasking function. Of course, individual values for each component of this metric would have provided more detailed information, but the coarse resolution of the currently available CLOCK function prevented measurement of individual values because of the large number of iterations needed to get a precise measurement. Iterating many times through a loop where tasks are created without being terminated causes the run-time system to thrash and prevents an accurate measurement. When higher resolution clocks are available, the source code of the test can easily be changed to time each individual part of the metric.

However, some additional information can be determined about the time for task activation. The test for measuring the composite of elaboration, activation, and termination is run for the two possible cases of task activation: entering the nondeclarative part of a parent block, and using the **new** allocator. In the first case, the task to be activated can be declared directly in the declarative part of a block, or it can be an object declared to be of a task type. To measure task activation time using the **new** allocator, an access type object is allocated that is a pointer to an object of a task type. The difference in the times provided by these three tests gives some insight into the relative efficiency of the two types of task activation.

Task Synchronization

Synchronizing tasks is important in multitasking. In Ada, synchronization is supported by the rendezvous mechanism, which allows tasks to pass information to one another at key points during their execution. To start the rendezvous involves at least two context switches: one to the run-time system, and then another to the acceptor if the latter is ready to accept the rendezvous. The run-time system must check to see if the acceptor is indeed ready to receive the rendezvous, and this adds to the overhead associated with context switches. If the overhead associated with a rendezvous is too great, the efficiency of execution in a multitasking environment will suffer.

The synchronization test measures the time needed to complete a rendezvous between a task and a procedure with no additional load. This method gives a lower bound on rendezvous time since no extraneous units of execution are compet-

ing for the CPU. This test is also repeated for the rendezvous mechanism where various numbers, types, and modes of parameters are passed.

Clock Function Overhead

In a real-time application, the CLOCK function provided in the CALENDAR package may be used extensively. The overhead associated with calling the CLOCK function can be an important contribution to the speed limit with which timed loops can be coded. The benchmark test in this case measures the overhead associated with a call to, and return from, the CLOCK function provided in the package CALENDAR. The method is essentially the same as the one used to measure the overhead associated with an entry and exit of a do-nothing subprogram in a separate package.

Arithmetic for Types TIME and DURATION

Dynamic computation of values of types TIME and DURATION is frequently a necessary component of real-time applications. An example is the difference between a call to the CLOCK function and a calculated TIME value, which is often used as the value in a **delay** statement. If the overhead involved in this computation is significant, the actual delay experienced will be somewhat longer than anticipated, and could be critical in the case of small delays.

The objective of the TIME and DURATION test is to measure the overhead associated with a call to and return from the "+" and "−" functions provided in the package CALENDAR. Times are measured for computations involving variables only and then both constants and variables. Although both "+" functions are essentially the same (only the order of parameters is reversed), both are tested, since a discrepancy in the time needed to complete the computation will occur if one of the functions is implemented as a call to the other.

Scheduling Considerations

Many real-time programs need to schedule tasks to execute at particular points in time, and to allow execution to switch among tasks. To provide programmers a mechanism for handling the former, Ada provides the **delay** statement. Switching execution among tasks can be achieved through a variety of mechanisms: The scheduler provided by the runtime system is entered at certain synchronization points in a program, at which time other tasks may be placed into execution; or the underlying system may implement a time slice mechanism. Great freedom is provided Ada implementors in realizing these mechanisms, and as a result, the schemes used can have a greater impact on the suitability of a particular implementation for real-time applications than the raw execution speed of many other constructs.

The principal issue involved, from a real-time perspective, is the mechanism by which tasks are placed into execution. The LRM states that the order of scheduling among tasks of equal priority, or among tasks of unstated priority, is undefined: Fair scheduling is presumed. Synchronization points are the beginning and end of task activations and rendezvous. In a system that does not implement priorities, these are the only points at which a user can be sure that the scheduler will be entered. The issue is determining when a task becomes eligible for execution after the expiration of a delay. An implementation may elect to check only for expiration of the delay periodically, at synchronization points, or in a variety of other ways.

Consider an embedded system in which the programmer has control over all nonsystem tasks to be executed, and a simple polling loop whose purpose is to receive messages from a network device and post them to a local mailbox. Although it would undoubtedly be desirable to have such a function interrupt driven, assume for this example that the underlying system precludes this possibility, hence the need for the polling loop. The basic loop, ignoring the need to allow other tasks to run, might reasonably have the following form:

```
loop
  if DEVICE_HAS_MESSAGE then
    RECEIVE(MESSAGE);
        --May be entry or procedure call          (L2)
    DEPOSIT(MESSAGE);
        --May be entry or procedure call
  end if;
end loop;
```

The problem is how to allow other tasks to occasionally obtain service from the CPU and still have the polling loop execute frequently enough that messages do not remain pending for long periods of time. The basic loop above must be modified to ensure that this occurs.

As a first strategy, suppose that a **delay** 0.05 statement is inserted before the **if** statement to provide an opportunity for other tasks to execute. One would expect that, if all tasks have equal or undefined priority, this strategy would allow other tasks the chance to run every time the message task runs, and, moreover, that the message task would have a chance to run in accordance with the underlying fair scheduling system. Further, if only the message task is ready to run, one would expect it to run approximately once every 50 ms. However, if, as is the case in some validated compiler systems, the expiration of this delay is only checked periodically,

say at 1-s intervals, to see if any delayed tasks are ready to be reactivated, the polling loop may only be executed once a second in spite of the fact that there are no other tasks ready to run. We call this type of scheduling *fixed-interval delay scheduling*. It can be performed quite independently from time slicing or other task scheduling that may be part of the same scheduling system.

If priorities are supported, one might also place a PRIORITY **pragma** before the loop to give the polling loop a higher priority and ensure that it will run in preference to other tasks, if ready. However, even in this case it is not clear when the implementation will check to determine if the delay has expired. This matter is presently under study by the Language Maintenance Committee, but because of the current uncertainty, it is wise to have a method for testing the scheduling algorithm used.

In order to develop many real-time Ada programs, it is clearly necessary to have supplemental information about the scheduling strategies used by an implementation. A method for determining the time slice interval was described on page 764. Next, we present techniques for determining the scheduling discipline related to delay expiration.

Delay and Scheduling Measurements

The test proposed here provides information regarding preemptive or fixed-interval scheduling. It is based on embedding a simple **delay** statement inside a loop that is executed a large number of times: For example,

```
T1 := CLOCK;
while I < N loop
  delay DEL;                    (L3)
  INCREMENT( I );
end loop;
T2 := CLOCK;
```

Obtaining the desired interpretations requires running this test for several different ranges of values of DEL. Typically, the proper value ranges will not be known a priori and may range over five orders of magnitude. The correct set of ranges must be determined empirically for each implementation. It will also generally be necessary to execute the test as the only process running on the CPU. Using this procedure, several useful interpretations can be obtained by plotting d(DEL) versus DEL where

$$d(DEL) = (T2 - T1)/N - TL \quad ,$$

and TL is the loop overhead time: That is, d(DEL) is the actual delay time achieved. Ideally, the points of this plot should lie on a straight line, with slope 1, as shown in Figure 1; deviations from this ideal will provide useful information about the scheduler.

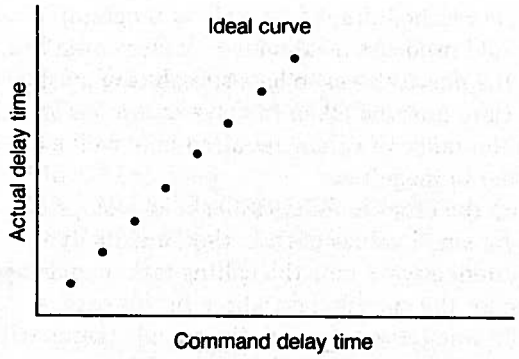

FIGURE 1. The Ideal Delay Curve

Minimum Delay Overhead. First, it is important to determine some information about the behavior of the scheduler for small values of DEL. Some implementations are smart enough to recognize situations in which the requested DEL is smaller than the overhead required by the **delay** function and simply do an immediate return to the calling unit. To study this, let T_δ be the time required to perform the **delay** operation, exclusive of any time the task is on a delay queue and the processor is performing work for another task. That is, T_δ is the overhead associated with delays. Typically, T_δ depends on DEL. For example, the overhead associated with returning to the calling program, if DEL is below some threshold, would be different from the overhead associated with placing the task in a delay queue.

To do the calculations, make a series of runs of loop L3 for increasing values of DEL, beginning with DEL = DURATION'SMALL, and generate the function given in Figure 1. If d(DEL) remains constant for small values of DEL (as shown in Figure 2), this suggests that, for DEL less than T_δ, the system does an immediate return to the calling program (or im-

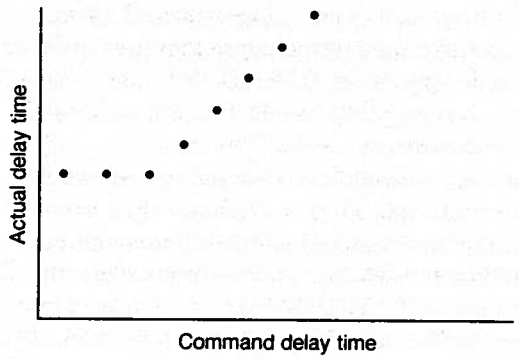

FIGURE 2. The Delay Curve for Small Values of DEL Showing Minimum Delay Overhead

mediate rescheduling of the calling program). The threshold used can be obtained by increasing DEL until the curve ceases to be a straight line of slope zero. Care must be taken in choosing values for DEL since the range of values required may well exceed an order of magnitude.

If, on the other hand, d(DEL) shows a slope of 1, even for small values of DEL, then it is likely that the system always puts the calling task on a delay queue for the specified duration. In this case, a straight line passed through the sample points will intercept the ordinate at the value of T_δ for small values of DEL. Unfortunately, this latter effect may be difficult to observe if scheduling is non-preemptive.

Fixed-Interval versus Preemptive Delay Scheduling. Next, we try to determine if fixed-interval delay scheduling, or true preemptive scheduling based on interrupts from a programmable clock, is being used. If, for DEL $> T_\delta$, the points of the plot lie on a straight line of slope 1, preemptive scheduling is indicated.

If the straight line with a slope of 1 is not achieved, it is suggestive that true preemptive scheduling is not being used. The plot is then likely to be a staircase function (i.e., if fixed-interval delay scheduling is being used). To see this, assume that only this task (loop L3) is executing and that, after the first iteration of the loop, the **delay** statement will be encountered very shortly after the expiration of one of the fixed scheduling intervals. If the DEL specified does not exactly reach the end of the next scheduling interval, sufficient extra delay will be inserted implicitly to reach the end of the scheduling interval. Thus, after the first loop, the actual delay will be approximately some multiple of the scheduling interval. If the scheduling interval is large compared to TL, then the size of the step in the plot will be approximately the interval of the scheduler as illustrated in Figure 3. Again, obtaining a sufficient set of values for d(DEL) is not entirely straightforward, as some compilers are known to have a scheduling interval more than five orders of magnitude larger than DURATION'SMALL. Some cleverness is required in selecting the values of DEL to use (e.g., a coarse–to–fine strategy).

There is one additional characteristic to a scheduling strategy that might complicate the interpretation. If the implementation does do preemptive scheduling, but with a time resolution element larger than DURATION'SMALL, a staircase plot will also result. Distinguishing between these cases can be difficult. If the measurement clock resolution τ is

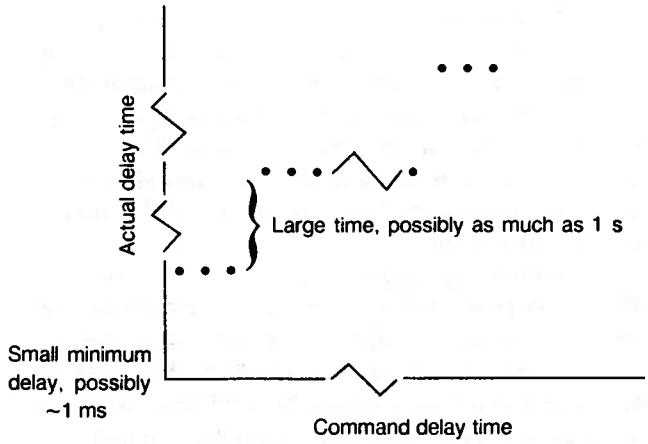

FIGURE 3. The Delay Curve for Fixed-Interval Scheduling

relatively small compared to $T1 - T2$ for $N = 1$, the two situations can be distinguished by rerunning the experiment for a fixed DEL with randomized starting times. In the case of true preemptive scheduling, $T2 - T1$ should remain relatively fixed, whereas, for fixed-interval delay scheduling, $T2 - T1$ will vary randomly, and the range of variation will correspond to the size of the interval of the scheduler.

Compensation for Minimum Delay Overhead. Finally, the situation where preemptive scheduling has been used and DURATION'SMALL is significantly less than T_δ provides us even further information. Although theoretically d(DEL) will be a straight line having slope 1 and passing through the origin, it will actually do so only if the system has compensated the delay time by T_δ. An offset of the line so that it does not pass through the origin indicates either no compensation for T_δ or incorrect compensation. More generally, due to the dependence of T_δ on DEL, the plot might be composed of several line segments, each of which could be examined as described above. If a fixed-interval delay scheduler has been used, this effect will be dominated by the extra delays introduced by the scheduler and will not be visible.

Although the data obtained in this test must be analyzed in several different ways, the data do allow a great deal of useful information to be determined about an implementation.

Memory Deallocation and Garbage Collection

Memory allocation and deallocation processes are often critical to the operation of real-time systems. Systems can fail when there is insufficient (virtual) memory available, when the allocation or deallocation times are too large, or when a deallocation pro-

cess (garbage collector) is implicitly called at times not under control of the applications program.

There are two reasons why insufficient memory failures might occur. First, there may just intrinsically be too little space available in the pool of storage from which allocations are made, although, for most systems, this problem will probably not occur. The second and more important reason, for real-time systems, is the fact that the LRM does not require an immediate return of deallocated storage to the storage pool. In fact, a validated compiler is available that does not return storage to the pool even if UNCHECKED_DEALLOCATION is called. Embedded systems are often expected to run for long periods of time, and although the total amount of storage in use at any one time may not be large, if deallocation does not take place, the system will eventually run out of storage unless the applications program takes over responsibility for storage allocation. It is preferable for storage deallocation in real-time systems to be under explicit control of the applications program. Some systems implicitly call a garbage collector—either periodically, or when the amount of allocated or unallocated storage reaches some threshold. Under these conditions, garbage collection can take a substantial amount of time, and unless it is run at the lowest possible priority (and priorities need not be supported), it can disrupt the operation of the system. Imagine a tight 1-ms control loop on an aircraft suddenly put into abeyance for a couple of seconds.

There are also interesting run-time or operating-system effects that are interesting to observe. In any virtual memory system, the amount of memory allocated can eventually reach the point where paging takes place. Both the amount of memory for which this occurs and the paging times required may be of interest. In one Unix system, it has been found that, when allocation storage approaches the virtual storage limit, overhead times of several seconds occur. (However, this is probably not a problem since the virtual size limit is so large that rarely, if ever, would one approach the limit.)

The basic idea behind building tests to measure these effects is to use the **new** allocator in a loop with various controls that determine whether or not it is possible for deallocation to take place. The second differencing technique described in "Basic Measurement Accuracy" on pages 762–763 can then be used to measure the relevant times.

In one test, a large array of pointers to a sizable array of data is declared. Each time the loop is executed, a pointer to a **new**ly allocated data array is placed in the pointer array, as shown below:

```
type INT_ARRAY is array(1..10, 1..10) of INTEGER;
type ARRAY_PTR is access INT_ARRAY;
PTR_ARRAY: array(1..MAX) of ARRAY_PTR;
TIME_ARRAY: array(1..MAX) of TIME;
begin                                              (L4)
  for I in 1..MAX loop
    PTR_ARRAY( I ) := new INT_ARRAY;
    TIME_ARRAY( I ) := CLOCK;
  end loop;
    :
```

This forces the storage acquired to be kept and not deallocated since the pointer to it remains throughout the run. By making the loop counter sufficiently high, more storage will eventually be requested than is available in the system, and the exception STORAGE_ERROR will be raised. A second difference analysis on the time array will yield the results on storage allocation and paging times.

A second test uses the same loop structure, but only two access variables. Each time the loop is executed, the content of one access variable is shifted to the second variable, and the newly acquired data are assigned to the first access variable, thus implicitly freeing the storage that was allocated two iterations prior to the current one. (This shifting is used to prevent the possibility that an optimizer will avoid the actual allocation of storage.) If the exception STORAGE_ERROR is also raised on this loop, lack of any implicit deallocation is indicated. If a garbage collector is implicitly called, this will be detected by the second difference analysis on the array of clock times.

The third test is similar to the second except that a call to UNCHECKED_DEALLOCATION is added to the loop to try to force deallocation. If the exception STORAGE_ERROR is still raised, either UNCHECKED_DEALLOCATION does not function properly, or there is some global limit on the amount of storage that can be allocated that is independent of the availability of storage to be allocated (a strange and unlikely occurrence).

Interrupt Response Time

Although interrupt response time is critical for many real-time embedded systems, techniques for measuring it are difficult to develop since hardware external to the CPU is generally involved. Second, the times that must be measured will be at substantively different points in the test program, and the use of iteration to improve accuracy of measurement (as in Equation 4) cannot be expected to work in this situation.

The first problem—the generation of the interrupt signal in a controlled and time-measurable fashion—

is approached by adding a parallel interface to the system being tested and writing a special driver for the interface that is directly accessible from the benchmark program. The output from the parallel interface is treated as a logic signal to cause an interrupt. The procedure for outputting a signal through this interface must be written to be directly callable, and hence time measurable, from the benchmark program, so that it does not have to go through the underlying run-time or operating system. Then, using techniques described earlier, it is possible to obtain an accurate measure of the time required to output a signal to the interface.

Two program segments are required for the benchmark. The first is a loop that repeatedly records the clock and outputs a signal to the parallel interface.

```
TIME_ARRAY: array(1..MAX) of TIME;
begin
  for I in 1..MAX loop
    TIME_ARRAY( I ) := CLOCK;                    (L5)
    SEND_SIGNAL;
        --to parallel interface & create interrupt
  end loop;
```

The second program segment is an interrupt handler that simply records the time at which it is invoked and returns from the interrupt. If possible, the interrupt handler should be set at a higher priority than the main loop.

The output procedure call and clock recording overhead can be calculated by the techniques described above. If T_{ov} represents this time, we can then calculate the average time difference between the times recorded in the main loop and the corresponding times recorded in the interrupt handler. If this average is denoted by T_{ave}, one can then calculate the interrupt response time as $T_{ave} - T_{ov}$.

RESULTS

In this section, we illustrate the application of the benchmarks in terms of their use with several compilers: Verdix Versions 4.06, 5.1, and 5.2 running with Unix 4.2 bsd on a VAX 11/780; DEC VAX Ada Version 1.1 running with Micro VMS 4.1 on a Microvax II; DEC VAX Ada Version 1.3 running with VMS 4.4 on a VAX 11/780; and Alsys Version 1.0 running with Aegis Version 9.2 on an Apollo DN 660. The specifics of the compiler/hardware combinations tested are given in Table I. All user and daemon processes were disabled (except for the swapper and page daemon, which can never be disabled). The tests described in "Measurement Techniques" were run to determine the operating-system overhead injected into the measurements for Unix on the VAX. Individually, the components to overhead required significantly less than the resolution of the time measurement, τ, making it difficult to get an accurate value for the overhead. Nonetheless, by examining the amount by which the string of zeros is shortened, we were able to obtain a crude estimate of the overhead. Using this approach, we estimated the overhead to be 5 percent. Owing to the coarseness of this estimate, we present the rest of the results without modifying them to reflect the operating-system overhead for time slicing.

The number of iterations used in the test and control loops was chosen to produce results that are theoretically accurate to the nearest 10th of a microsecond or 10th of a millisecond (except where noted), depending on the size of the quantity being measured. The results were very repeatable. Raw control and test results were usually repeatable to within 0.1 or 0.2 μs (per iteration) for tests with similar target accuracies. This allowed us to see the effects of single instruction differences between two different situations and exposed a number of interesting implementation variations.

We found that similar but not identical situations (e.g., passing one parameter versus passing several parameters) resulted in slightly different code sequences for some compilers. We even found positional dependencies in which the timing varied among identical functions on the basis of the relative position of units within a package, or their position relative to double word boundaries in memory (re-

TABLE I. The Compiler/Hardware Combinations

Compiler	Machine	Operating system
Verdix 4.06	Vax 11/780, 4M real memory	Unix bsd 4.2
Verdix 5.1	Vax 11/780, 4M real memory	Unix bsd 4.2
Verdix 5.2	Vax 11/780, 4M real memory	Unix bsd 4.2
Alsys 1.0	Apollo DN660, 4M real memory	Aegis Version 9.2
DEC VAX Ada, V.1.1	DEC Microvax II, 5M real memory	Micro VMS 4.1
DEC VAX Ada, V.1.3	DEC VAX 11/780	VMS 4.4

TABLE II. Subprogram Overhead

Compiler	Interpackage (μs)	Intrapackage (μs)	In-line (μs)	Generic interpackage (μs)	Generic intrapackage (μs)
Procedure calls, no arguments					
Verdix 4.06	17.7	27.7	28.6	25.9	30.5
Verdix 5.1	17.6	16.8	0.4	18.5	17.6
Verdix 5.2	18.0	17.0	0.0	18.0	20.0
Alsys 1.0	14.0	12.0	31.0	8.0	27.0
DEC VAX Ada, V.1.1	46.1	x[a]	3.0	45.9	x[a]
DEC VAX Ada, V.1.3	27.0	x[a]	0.0	15.0	x[a]

Compiler	In (μs)	Out (μs)	In out (μs)
Additional overhead per integer argument			
Verdix 4.06	~1.5	~3.0	~3.0
Verdix 5.1	~1.5	~3.0	~3.0
Verdix 5.2	~1.5	~3.0	~3.0
Alsys 1.0	~4.2	~2.8	~4.7
DEC VAX Ada, V.1.1	~1.3	~3.0+	~6.0+
DEC VAX Ada, V.1.3	~1.5	~3.0+	~6.0+

[a] Compiler INLINEd the call reducing value to zero. DEC has supplied a value of 15.1 μs for this call.

lated to the number of memory fetches required). With the assistance of some of the compiler vendors, we tracked down exactly what was happening in a number of such cases to be sure that our benchmarks were correct. We will describe some of these below as illustrations of the differences that can occur.

As a result of such minor variations, it is difficult to place meaning on results any closer than a couple of microseconds (even though theoretically more accurate results have been obtained) for two reasons: First, the number of special cases to track down is sufficiently large as to require a very large effort to be comprehensive, and second, even if one did track down each situation completely, there would be so many separate cases to report that one could not reasonably attempt to use all of the data.

The test results are summarized in Tables II–VII, where all values are reported in microseconds except where noted. A complete listing of all results is given in [4].

Subprogram Overhead

The times resulting from procedure calls of various kinds are given in Table II, which contains several surprises. First, it is evident that simply checking one kind of procedure call is inadequate. For some compilers, the differences among different kinds of calls—generic, nongeneric, intra- or interpackage—can be as much as two to one. Detailed investigation of DEC VAX compiler outputs showed that there were differences in certain elaboration and stack checks between the generic and nongeneric versions of the code.

A second characteristic of subprogram overhead that is not obvious from the table is the effect of code optimization. The DEC compiler, for instance, will in-line procedures for small procedure sizes automatically as a time optimization, even if INLINE is not used. Although this improves performance substantially, it makes it difficult to test procedure calling time and raises some question about the interpretation of the results. The numbers not available for procedure calls in Table II indicate circumstances in which the compiler INLINEd the test procedure, reducing the time to near zero.

As a second illustration of minor code differences, consider the procedure call times with 1 or 10 integer arguments (not shown in detail in Table II due to the size of the data). For a single integer argument, the calling time was less for **in out** mode parameters than for **out** mode parameters. This relative timing was reversed when 10 parameters were passed. The reason is that the DEC peephole optimizer could see that a single **in out** formal did not receive an assignment (in our benchmark) and therefore optimized the exiting assignment out of the code, whereas, for 10 parameters in the parameter list, the window was too small for that observation to be made, and the exiting assignment was done for all parameters. That

TABLE III. Dynamic Storage Allocation

Compiler	Dynamically bounded arrays[a] in declarative region: 1-D array (μs)	2-D array (μs)	3-D array (μs)
Verdix 4.06	31	46	54
Verdix 5.1	143	149	161
Verdix 5.2	19	31–32	43–46
Alsys 1.0	28–41	74–84	145–168
DEC VAX Ada, V.1.1	9–18	22–25	37–38
DEC VAX Ada, V.1.3	13–18	21–31	46–48

Compiler	Dynamically bounded arrays[b] allocated via *new*: 1-D array (μs)	2-D array (μs)	3-D array (μs)
Verdix 4.06	221–284	309–1200	326–x[c]
Verdix 5.1	200–260	280–1140	300–3370
Verdix 5.2	220–280	290–1300	300–3350
Alsys 1.0	2249–2185	2191–2217	2300–2334
DEC VAX Ada, V.1.1	410–450	430–870	490–4830
DEC VAX Ada, V.1.3	290–300	280–300	370

Compiler	Fixed-length objects (small, no arrays) located via *new*
Verdix 4.06	133–300
Verdix 5.1	227–270
Verdix 5.2	130–239
Alsys 1.0	1963–1985
DEC VAX Ada, V.1.1	310–510
DEC VAX Ada, V.1.3	250

[a] Integer arrays with range 1 along each dimension.
[b] Two tests each, integer arrays with ranges 1 and 10 along each dimension.
[c] Storage errors resulted when we attempted to allocate larger amounts of storage.

optimization was not performed for the **out** mode case.

The per argument times associated with procedure calls were checked for lists of 1, 10, and 100 arguments of INTEGER and ENUMERATION types, except for the Alsys and VAX Version 1.3 compilers, which would not handle argument lists of length 100. The differences in times among the modes seem to indicate copying associated with pass by value and initialization of variables. Variations also occurred in the number of registers used (and therefore saved and restored) as a function of the number of parameters passed. The "+" in the per argument table indicates that a fraction of a microsecond was added to each argument passed, depending on the number of registers used.

Although we did obtain repeatable results with the Alsys compiler, the results did not fit a linear formula well and are thus not reported that way. The values were in the range of 4–7 μs per argument.

Dynamic Allocation of Objects

The memory allocation tests given in Table III are divided into two categories: allocations performed in a declarative region on entering a procedure, and allocations performed via the **new** allocator.

The time required for fixed-size storage allocation in a declarative region was small (a few microseconds) and roughly constant for each of the compilers, and so was not shown in the table. The time required for dynamically bounded arrays varied approximately as a linear function of the number of dimensions, which was expected (considering the formulas typically used for computing array dope vectors). The times were in the 10–20 μs per dimension range for all compilers except the Verdix 5.1 and Alsys compilers, where the times were appreciably larger. All the ranges used in these tests were kept small to avoid other storage effects, like allocating from the heap for objects above some size threshold.

Two significant effects were discovered that had

to be taken into account in order to obtain useful results in dynamic allocation via the **new** operator. First, for the Verdix 4.06 compiler, problems arose with the underlying memory management mechanism. Since this compiler version never deallocated storage, as the amount of storage allocated across a large number of iterations began to grow, the operating system began to swap memory pages onto disk, which was sufficient to distort the test results. To eliminate this difficulty, a sequence of pretests was run to determine the number of iterations that could be included in the test before paging became a significant problem. The tests were then run with this number of iterations. This reduced precision somewhat, but useful results were still obtained. Verdix Versions 5.1 and higher did deallocate storage, which, while it eliminated the paging problem, did increase slightly the storage times recorded.

Second, most of the compilers used a multilevel storage allocation scheme. Small objects were allocated from some locally held storage pool, whereas for larger objects calls were made to the underlying system for more storage. The latter situation was quite evident since the larger objects typically required nearly an order of magnitude larger time than objects allocated from the local pool. To make these results evident, the dynamic storage requests via **new** were run several times with object sizes ranging from 4 to 4000 bytes. The wide range of times shown in Table III simply reflects the fact that small objects were allocated locally while large objects required a system call.

The multilevel nature of dynamic storage allocation was also found in the CLOCK function although it was not easy to detect. The Verdix CLOCK function dynamically allocates a record each time it is called. Although the time needed to allocate this record from the local pool is usually only a few 10s of microseconds, every once in a while the local pool becomes exhausted, and a system call must be made to obtain more storage; when this happens, the time needed to obtain a new chunk of storage is on the order of 3 ms. Thus, the time to allocate any one object can be quite variable. The possibility of a CLOCK call occasionally taking a long time due to the need to acquire more storage can have a devastating effect on some real-time programs, as CLOCK will be used in many, if not most, real-time scheduling loops. Since the system call for more storage does not happen very often, it will be difficult to isolate the problem. Consequently, it is important to identify all implementation-supplied procedures or functions that allocate storage.

Exceptions

The exception-handling tests are divided into two sets. In the first set, an exception is raised within a declare block, and the exception handled by a handler at the end of the block. In the second set, an exception is raised from within a procedure that does not have an explicit handler. The exception is then propagated to the calling block, which handles the exception at the end of the block from which the procedure was called. Exceptions were raised by three methods—explicitly with the **raise** statement, by violating a subtype range, and through INTEGER overflow.

The results of the exceptions tests are shown in Table IV. In general, the compilers take little or no time for exceptions that are not raised, which is important for real-time applications. However, all the exception-handling times are significantly longer than would be required for condition testing and subprogram calls. When very fast response is required, users may find it necessary to explicitly handle exceptional situations in the body of their programs rather than relying on the Ada exception mechanism. The much larger times associated with implicitly raising NUMERIC_ERROR are associated with the fact that this kind of error is first trapped

TABLE IV. Exception Handling

Compiler	User defined, not raised		Numeric error, implicitly raised in procedure (ms)	Other exceptions	
	Block (μs)	Procedure (μs)		Handled in block	Propagated to calling procedure
Verdix 4.06	0	0	2.47	345–402 μs	614–671 μs
Verdix 5.1	0	1	2.55	315–372 μs	544–613 μs
Verdix 5.2	0	1	2.72	396–448 μs	718–783 μs
Alsys 1.0	0	0	17.95	8.80–9.80 ms	19–20 ms
DEC VAX Ada, V.1.1	4	16	0.89	667–836 μs	736–894 μs
DEC VAX Ada, V.1.3	3	12	0.60	414–541 μs	482–619 μs

TABLE V. Tasking Times

Compiler	Rendezvous (ms)	Task elaborate, activate, terminate (ms)
Verdix 4.06	3.50	19.6
Verdix 5.1	3.40	20.4
Verdix 5.2	0.82–0.89	3.6
Alsys 1.0	9.55	14.2
DEC VAX Ada, V.1.1	1.85	8.2
DEC VAX Ada, V.1.3	1.10	6.6

by the operating system, which then passes control back to the exception handler. In an embedded system with a dedicated real-time operating system, this time could be significantly less than occurred in our tests on a time-shared system.

Task Elaboration, Activation, and Termination

This test was run for the three different types of task elaboration and activation explained on page 767. The task elaboration, activation, and termination times for the compilers tested are given in Table V. For each individual compiler, the differences between elaboration and activation in a declarative region versus the **new** operator did not differ by more than 15 percent and thus are not reported separately. Table V shows that efficient techniques for task elaboration, activation, and termination are possible.

Task Synchronization

The test here was rather straightforward: It involved entering a block where a task is activated and a subprogram is called that executes a rendezvous with that task repeatedly in a loop. The control for this test is of the same structure, except that the loop is iterated with no rendezvous. As can be seen in Table V, the rendezvous times varied significantly, again indicating that, as development continues on successive versions of compilers, the rendezvous times can be decreased. Entry calls with parameters showed that the additional time needed to pass parameters was negligible.

Clock Function Overhead and Resolution

Table VI shows the overhead associated with the CLOCK function. The numbers reported are averages obtained over several test runs and show a large variation in the length of time required by the different compilers. The large increase in overhead required by the Verdix 5.1 and 5.2 compilers is due to a change in the data structure for objects of type TIME, and an increase in the number of procedure and function calls within the CLOCK function. Unix system routines are called by CLOCK to get the time and compensate for the time zone. Daylight saving time is also taken into account, and the time is normalized with respect to Greenwich meridian time. Since TIME objects are represented as Julian days and seconds, an Ada function in the CALENDAR package is also called to compute the Julian day. We were able to determine this information about the CLOCK function by examining the source code of the body of the CALENDAR package. Although useful for some applications, these extensive computations are too time expensive for many real-time applications, and some additional clocklike function will be required for real-time applications. A CLOCK resolution of 10 ms is marginal for many real-time applications.

Arithmetic for Types TIME and DURATION

The TIME math tests measure the overhead involved in addition and subtraction operations involving types TIME and DURATION. All possible combinations involving variables and constants of each type are tested; the results are presented in Table VII. It appears that constant expressions are evaluated at compile time in all the compilers. The difference of more than an order of magnitude between operations on type TIME and type DURATION is probably due to the representation of TIME as a record, whereas DURATION is fixed point. The variation in results between the different versions of

TABLE VI. Timing and Scheduling

Compiler	Clock call (μs)	Clock resolution	Delay scheduling method	Effective delay resolution
Verdix 4.06	570	10 ms	Fixed interval	Variable 10 ms–1 s
Verdix 5.1	3550	10 ms	Fixed interval	Variable 10 ms–1 s
Verdix 5.2	3644	10 ms	Preemptive	10 ms
Alsys 1.0	1500	1 s	?	1 s
DEC VAX Ada, V.1.1	95	10 ms	Preemptive	10 ms
DEC VAX Ada, V.1.3	89	10 ms	Preemptive	10 ms

TABLE VII. Time

Compiler	TIME's only (μs)	DURATION's only (μs)	DURATION := TIME − TIME; (μs)
Verdix 4.06	188–241	7.2–7.6	111
Verdix 5.1	716–812	6.3	50
Verdix 5.2	816–889	6.0	75
Alsys 1.0	88–105	1.0–2.0	189
DEC VAX Ada, V.1.1	98–109	x*	118.0
DEC VAX Ada, V.1.3	91–94	1.0	94

* Reliable data unavailable.

the Verdix compiler for expressions involving type TIME is due to a change in the record used to represent TIME.

Delay and Scheduling Measurements

The results of measuring the time elapsed during the execution of a **delay** statement appear in Table VI. For the Verdix 4.06 compiler, a minimum delay value of 1.4 ms was detected. This delay occurred for requested delays between zero and slightly less than 1 ms (actually, the upper bound is 16 times DURATION'SMALL, the greatest model number less than 1 ms). This value corresponds to the part of the curve before the jump in Figure 3. For the Verdix 5.2 and DEC compilers, the minimum actual delay was 10 ms, while for the Alsys compiler it was 1 s.

The actual delay values in other cases were more difficult to isolate, due to the nature of the scheduling systems. Verdix Versions 4.06 and 5.1 use fixed-interval delay scheduling with a delay value of 1 s. Thus, for a requested delay of 1 ms or greater, the actual delay was for the remainder of the 1-s time slice in which the delay expired. Since it is impossible to see this effect when a large number of iterations are run, the test was run repeatedly with the loop executed only once on each test. A delay generated by executing a statement a random number of times was inserted before the **delay** statement to vary the value remaining in the time slice. This procedure confirmed that requested delays between 1 ms and less than 10 ms resulted in actual delays between 10 ms and 1.01 s, or the value remaining in the 1-s time slice plus 10 ms. As the requested delay was increased, the staircase function of Figure 3 was obtained, with a 1.01-s step size. The extra 0.01 s corresponds to one clock resolution time and appears to be time spent in the scheduler before the basic 1-s time slice is reset.

Both the Verdix Version 5.2 and the DEC compiler used preemptive scheduling with a time resolution of 10 ms. Owing to the 1-s time resolution of the Alsys compiler, it was neither practical nor useful to test its scheduling algorithms further.

Storage Deallocation and Garbage Collection

The storage deallocation tests provided insight into the type of deallocation facilities provided for objects declared dynamically with the **new** allocator. The object used for allocation throughout this test was a one-dimensional array consisting of 1000 INTEGERs. The size of the virtual memory space available is approximately 32 Mbytes, the limit imposed by the operating system and the size at which STORAGE_ERROR should be raised by loop L4.

By modifying the test loop to use only two access variables, instead of the array of access variables in L4, we found that the Verdix Version 4.06 run-time system does not perform garbage collection, since STORAGE_ERROR was still raised at the same point. Further, by explicitly calling UNCHECKED_DEALLOCATION after every allocation and observing that STORAGE_ERROR was still raised at the same point, we concluded that the UNCHECKED_DEALLOCATION procedure does not reclaim storage in that version of the compiler.

Version 5.1 and 5.2 compilers also do not perform garbage collection, but UNCHECKED_DEALLOCATION does reclaim storage for scalar types, records, strings, and statically bounded array types. Storage is not reclaimed for unconstrained array types. Owing to improper setting of system parameters on the Microvax, the DEC Ada Version 1.1 tests (performed by a third party) were ill behaved for large amounts of storage allocation, and the tests were not performed on this version.

SUMMARY AND CONCLUSION

This article has presented a series of benchmarks to test the real-time performance of an Ada compiler and run-time system, together with a set of analysis tools to aid in the interpretation of test results. To obtain accurate results, the tests should be run as the sole application on the machine being used, with as many system daemons disabled as possible. To verify the quality of the environment before running the tests, a simple procedure of repeatedly reading the system clock and analyzing the results to identify the frequency and size of operating-system activity should be performed.

Although the benchmarks are designed for testing real-time performance, the only Ada systems available to us at the time of development were intended for time-shared, not real-time, use. Time-shared systems often place less emphasis on real-time perfor-

mance than on general program development and execution support, and the results of our tests bore this out. However, by the same token, the results point to areas where users should expect significant performance improvements in systems intended for real-time applications, including improved performance of the task scheduler, the incorporation of **pragma** INLINE, improved storage management facilities, higher speed operations with respect to TIME, and a reduction in tasking and CLOCK overhead.

There are a small number of real-time relevant tests that we were not able to perform on the systems available to us (e.g., the interrupt response time and the behavior of the system with respect to task scheduling upon I/O requests). However, a test has been developed for interrupt response time, and the time-shared operating system determines the behavior of I/O at a level above the tasking level of the Ada program. Further work is required in these areas when suitable testing facilities are available.

Finally, based on our experience in developing these benchmarks, we argue that since so many implementation-dependent variations are validatable, it is not safe, in our opinion, to use an Ada compiler for real-time applications without first checking it with performance evaluation tools. Time management, scheduling, and memory management can have validated implementations that will devastate a real-time application. Moreover, since real-time performance evaluation is difficult due to the great variety of implementation dependencies allowed, it typically requires interpretation and benchmark changes for each individual compiler tested. And real-time performance evaluation is really only meaningful for dedicated embedded systems.

Acknowledgments. The authors wish to thank Chuck Antonelli for sharing his knowledge of the Unix operating system and for his help in obtaining and interpreting direct time readings from Unix. We also thank Ron Theriault, Jarir Chaar, and Sue Hsieh for numerous late nights in helping run the benchmark tests, and Bill Meier of Digital Equipment Corporation for running our benchmarks on a VAX 11/780 and assisting us in deciphering some of the minor anomalies that occurred.

REFERENCES

1. Ada Joint Program Office. Ada programming language (ANSI/MIL-STD-1815A). OUSD(R&D), Dept. of Defense, Ada Joint Program Office, Washington, D.C., Jan. 1983. The official reference manual for the Ada programming language.
2. Barnes, J.G.P. *Programming in Ada.* Addison-Wesley, Reading, Mass., 1984. A textbook on Ada written by one of the original design teams; among other things, it gives examples of using Ada for real-time control loops.
3. Bassman, M.J., Fisher, G.A., Jr., and Gargaro, A. An approach for evaluating the performance efficiency of Ada compilers. In *Ada in Use, Proceedings of the Ada International Conference* (Paris, France, May 14–16). ACM, New York, 1985, pp. 72–84. Presents techniques for measuring specific features of a language and blocking code optimizers.
4. Clapp, R.M., Duchesneau, L., Volz, R.A., Mudge, T.N., and Schultze, T. Toward real-time performance benchmarks for Ada. RSD-TR-6-86, Electrical Engineering and Computer Science Dept., Univ. of Michigan, Ann Arbor, Jan. 1986, pp. 1–25. Provides a more complete listing of the test results.
5. Curnow, H.J., and Wichmann, B.A. A synthetic benchmark. *Comput. J. 19*, 1 (Feb. 1976), 43–49. The classic set of benchmarks oriented toward scientific programs for comparing the computational performance of computers and/or languages. Together, these benchmarks form one synthetic benchmark.
6. Jalics, P.J. Comparative performance of Cobol vs PL/1 programs. In *Computer Performance Evaluation Users Group 16th Meeting* (Orlando, Fla., Oct. 20–23). Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C., 1980, pp. 53–59. Discussions and results for several benchmarks that are designed to measure language features in Cobol and PL/1.
7. Shimasaki, M., Fukaya, S., Ikeda, K., and Kiyono, T. An analysis of Pascal programs in compiler writing. *Softw. Pract. Exper. 10*, 2 (Feb. 1980), 149–157. Static statement frequencies and dynamic frequencies of p-code instructions are given for several Pascal compilers written in Pascal.
8. Squire, J. Performance issues workshop. In *ACM SIGADA Users Committee Performance Issues Working Group* (Baltimore, Md., July 15–16). ACM, New York, 1985. The first in a series of workshops that will bring together and run a set of benchmark programs comparing Ada compilers.
9. Weicker, R.P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM 27*, 10 (Oct. 1984), 1013–1030. Presents a synthetic benchmark program, emphasizing use of records and pointer data types, that is based on statement frequency, number of parameters, operand types, and operand locality of a very large set of programs written in various high-level languages.
10. Zeigler, S.F., and Weiker, R.P. Ada language statistics for the iMAX 432 operating system. *Ada Lett. 2*, 6 (May 1983), 63–67. Reports statistics on the percentage of Ada code in various categories, such as subprogram calls.

CR Categories and Subject Descriptors: C.4 [**Performance of Systems**]; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.m [**Programming Languages**]: Miscellaneous; J.7 [**Computers in Other Systems**]: *real time*; K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*benchmarks*

General Terms: Languages, Management, Measurement, Performance, Verification

Additional Key Words and Phrases: Ada compiler evaluation, benchmarks, real-time benchmarks, real-time performance evaluation, real-time systems

Received 12/85; revised 3/86; accepted 5/86

Authors' Present Addresses: Russell M. Clapp, Louis Duchesneau, Richard A. Volz, Trevor N. Mudge, and Timothy Schultze, The Robotics Research Laboratory, The College of Engineering, The University of Michigan, Ann Arbor, MI 48109.