RSD-TR-12-86

TOWARD REAL-TIME PERFORMANCE BENCHMARKS FOR ADA

Russell M. Clapp

Louis Duchesneau

Richard A. Volz

Trevor N. Mudge

Timothy Schultze

July 1986

Robot Systems Division

Center for Research on Integrated Manufacturing

College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109 USA

RSD-TR-12-86

TOWARD REAL-TIME PERFORMANCE BENCHMARKS FOR ADA[1,2]

(Second Edition)

Russell M. Clapp
Louis Duchesneau
Richard A. Volz
Trevor N. Mudge
Timothy Schultze

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

July 1986

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

[1]Ada is a registered trademark of the Department of Defense.

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

TABLE OF CONTENTS

TOWARD REAL-TIME PERFORMANCE BENCHMARKS FOR ADA[1,2]

by

Russell M. Clapp
Louis Duchesneau
Richard A. Volz
Trevor N. Mudge
Timothy Schultze

The Robotics Research Laboratory
The College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

This paper addresses the issue of real-time performance measurements for the Ada programming language through the use of benchmarks. First, the Ada notion of time is examined and a set of basic measurement techniques are developed. Then a set of Ada language features believed to be important for real-time performance are presented and specific measurement methods discussed. In addition, other important time related features which are not explicitly part of the language but are part of the run-time system are also identified and measurement techniques developed. The measurement techniques are applied to the language and run-time system features and the results are presented.

1. Introduction

"Ada is the result of a collective effort to design a common language for programming large scale and real-time systems." So states the foreword to the Ada Language Reference Manual [1]. Examples of real-time systems include the avionic system in an airplane, the system that controls and commands a robot, and even the controller for a video game. The common denominator among these applications is the need to meet a variety of real-time constraints. While Ada was intended for such applications, there is nothing in the Language Reference Manual (LRM) which ensures that Ada programs, regardless of processor speed, will have the performance 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 was not intended to specify the size or the speed of their object code, or the precise nature of their task scheduling mechanisms, all of which are critical to real-time performance. The language contains mechanisms intended for real-time applications but

[1]Ada is a registered trademark of the Department of Defense.

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

leaves performance issues to supplemental measurement. This paper addresses the issue of real-time performance measurement, particularly the issues of time measurement and scheduling for which adequate requirements for real-time applications are *not* explicitly stated in the LRM.

Benchmarks provide a direct way to measure performance. This paper will explore the design and use of a set of benchmarks suitable for measuring the real-time performance of the code produced by an Ada compiler. Benchmarking can be approached in two ways:

- develop a composite benchmark, such as the Whetstone or the Dhrystone [2, 3], or,
- develop a set of benchmarks, each measuring the performance of a specific feature of the implementation, [4].

The former is easier to apply, but no single composite can capture all of the information required for even a modest spectrum of real-time applications. Moreover, detailed knowledge of the performance of individual features is often required for applications planning. In addition, such knowledge will be useful in understanding the relation between real-time performance, language constructs, and compiler implementation. Therefore, our approach will concentrate on techniques for measuring the performance of individual language features.

The development of benchmarks to measure the performance of individual language features involves a number of complex operations, including:

- isolation of the feature to be measured;
- achieving measurement accuracy;
- achieving measurement repeatability;
- elimination of underlying operating system interference from
  - time slicing,
  - daemons,
  - paging.

Each of these operations is considered in this paper. In addition to the performance of individual language features there are other real-time performance measurements that are associated with the run-time system. These include measurements of the scheduling and storage management algorithms.

This paper focuses on features from the language and run-time system believed to be important for real-time performance, concentrating not only on the benchmarks, but 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 [5]. Most of the benchmark tests presented in this paper were contributed to that effort during the summer of 1985. The remainder of the tests, those developed during Fall of 1985, were contributed in Winter of 1986.

The development and interpretation of measurement techniques for real-time programming is based upon the Ada notion of time. Section 2 reviews this notion. Section 3 presents techniques for achieving basic measurement accuracy, isolating the features to be measured, and determining the interference of operating system functions. Section 4 presents the set of features believed to be important for real-time performance, discusses why they are considered to be important, and describes the measurements to be made by the benchmark. Particular focus is given to scheduling operations and time measurements. Section 5 then 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 DN 660. It should be noted that these versions of the compilers are intended for time-shared use, *not* for real-time applications. 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 give an indication of the areas in which users should look for improvements in cross-compilers intended for real-time applications.

2. 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. These include:

(1) The data type TIME, objects of which type are used to hold an internal representation of an absolute point in time.

(2) The data type DURATION, objects of which type are used to hold values for intervals of time.

(3) A predefined package, CALENDAR, which provides functions to perform arithmetic on objects of type TIME or DURATION.

(4) A predefined function, CLOCK, which returns a value of type TIME corresponding to the current time.

(5) DURATION'SMALL, which gives an indication of the smallest interval of time which can be represented in a program. It is required to be less than or equal to 20 milliseconds, with a recommendation that it be as small as 50 microseconds.

(6) The value SYSTEM.TICK, which is defined as the basic system unit of time.

(7) 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. Those of the last four warrant some discussion.

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. They do not imply in any way actual performance of a system for time measurements or scheduling. 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 microseconds, while the timer resolutions are 10 milliseconds and 1 second respectively.

The CLOCK function generally presumes an underlying clock or timer which 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 that it means 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 which are not identified in the LRM. For example, some validated Ada implementations frequently insert sizable delays in conjunction with the delay statement which are neither directly specified by the programmer, nor caused by system load, but are present simply for convenience in the implementation of the complier and run-time system. Parameters in this category will be identified in the discussion that follows and techniques for measuring them will be presented.

3. Measurement Techniques

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

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

- isolation of the feature to be measured and avoidance of compiler optimizations which would invalidate the measurement;
- obtaining sufficient accuracy in the measurement;
- avoidance of operating system distortions;
- obtaining repeatable results.

These issues are dealt with in the subsections below.

3.1. Isolation of Features

The basic technique for isolating a specific feature to be measured from other features of the language is to use two execution frames, a control loop and a test loop which differ only by the feature whose execution time is to be measured. Thus, a difference of execution times between the control loop and the test loop theoretically 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, therefore, must utilize techniques to thwart code optimizers.

The key to avoiding these problems is to not 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 removing the execution of the feature being tested from the loop or eliminating the loop entirely from the control loop which does not contain the feature. To ensure that these problems do not happen, control functions are inserted into both loops and the feature being measured is placed in a subprogram called from a library unit [6]. Again, if the bodies of these subprograms are compiled separately, and after the benchmark itself, there is no way for a compiler 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.

3.2. Basic Measurement Accuracy

Knowledge of both the resolution of a time measurement and the variability of the time needed to make a time measurement are required to determine the number of iterations needed to obtain a parameter measurement within a given tolerance. Let r be 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

$$\left\lfloor \frac{t + r_c \pm r_v}{r} \right\rfloor \cdot r \ , \tag{1}$$

where $\lfloor x \rfloor$ is the "floor" function (the largest integer less than or equal to x), r_c is the nominal time required to perform the CLOCK function, and r_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, r_c will subtract out of the equations to be developed and can be ignored. It is assumed in all of the equations that follow that r_v is small in comparison to r and can also be ignored. In any application, however, this assumption must be verified. One of the tests described in the Sec. 4 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

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

where

$$0 \leq \delta < 1.$$

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 accuracy desired. However, one must be aware of cumulative error buildup, and if T_0 is obtained by a similar type of measurement, one must increase N for both measurements.

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

Let the time to complete one execution of the loop be

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

Without loss of generality consider that the first execution of the loop begins at time zero. Then the time at the end of the k th 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,

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

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

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

In the second difference of the times measured by the CLOCK function the $n \cdot \tau$ in (8) will subtract out to yield,

$$H(\delta, k) = \Delta T^m(k) - \Delta T^m(k-1) = \{\lfloor (k+1) \cdot \delta \rfloor - 2 \cdot \lfloor k \cdot \delta \rfloor + \lfloor (k-1) \cdot \delta \rfloor\} \cdot \tau \tag{10}$$

Now, if $H(\delta, k)$ is plotted as a continuous function of k with δ as some fixed constant in the interval [0,1), the periodic waveform of Fig. 1 is obtained with $1/\delta$ as the period.

The value x in Fig. 1 is the interval of k for each period where $H(\delta, k)$ is equal to r or $-r$. The function is equal to zero when it is not r or $-r$, so for each period, this interval of k is $y = 1/\delta - 2 \cdot x$. The value of y for $H(\delta, k)$ is dependent on the value of δ. To determine y consider the three floor function terms that comprise $H(\delta, k)$. Observe that, because of the coefficient of -2 on the second term, the function is equal to zero only when all three terms evaluate to the same value or three consecutive integers. This implies that $H(\delta, k)$ is equal to r or $-r$ only when two of the terms are equal and the third term is one greater or one less. Rewriting the three terms in $H(\delta, k)$ as follows:

$$\lfloor k \cdot \delta + \delta \rfloor, \quad \lfloor k \cdot \delta \rfloor, \quad \lfloor k \cdot \delta - \delta \rfloor \quad , \tag{10.1}$$

it is easy to see that for all three terms to be equal, it is necessary that $\delta < 1/2$. Also, it can be observed that for all three terms to have different values it is necessary that $\delta > 1/2$. If $\delta = 1/2$, neither case applies, and two of the three terms are always equal. Therefore, there are three cases to consider when determining the value of y. The simple case is when $\delta = 1/2$. Since $H(\delta, k)$ is never equal to zero in this case, $y = 1/\delta - 2 \cdot x = 0$, implying that $x = 1/2 \cdot \delta = 1$.

The second case to consider is when $\delta < 1/2$. To find the interval of k for which all three terms in (10.1) are equal, let $k \cdot \delta - \delta$ equal some integer m. Then, since $\delta < 1/2$, $\lfloor k \cdot \delta + \delta \rfloor = m$ and $\lfloor k \cdot \delta \rfloor = m$. To find y let $(k + y) \cdot \delta + \delta = m + 1$ i.e. the point where the function steps up by r. When this is true, $\lfloor (k + y) \cdot \delta - \delta \rfloor = m$ and $\lfloor (k + y) \cdot \delta \rfloor = m$ and $H(\delta, k)$ is no longer equal to zero. So,

$$(k + y) \cdot \delta + \delta = m + 1 \tag{10.2}$$

$$\therefore \quad (k \cdot \delta - \delta) + y \cdot \delta = m + 1 - 2 \cdot \delta \tag{10.3}$$

$$\text{i.e.,} \quad m + y \cdot \delta = m + 1 - 2 \cdot \delta \tag{10.4}$$

$$\therefore \quad y = 1/\delta - 2 \tag{10.5}$$

and then $y = 1/\delta - 2 \cdot x = 1/\delta - 2$ implying that $x = 1$.

A similar analysis is performed for the third case when $\delta > 1/2$. In this case $H(\delta, k)$ is zero when the terms of (10.1) evaluate to three consecutive integers. Let $k \cdot \delta + \delta = m$ so that $\lfloor k \cdot \delta \rfloor = m - 1$ and $\lfloor k \cdot \delta - \delta \rfloor = m - 2$. Again, let $y = 1/\delta - 2 \cdot x$ and then let $(k + y) \cdot \delta - \delta = m - 1$. When this is true, $\lfloor (k + y) \cdot \delta \rfloor = m - 1$ and $\lfloor (k + y) \cdot \delta + \delta \rfloor = m$ and the function is no longer equal to zero. So

$$(k + y) \cdot \delta - \delta = m - 1 \tag{10.6}$$

$$\therefore \quad (k \cdot \delta + \delta) + y \cdot \delta = m - 1 + 2 \cdot \delta \tag{10.7}$$

$$\text{i.e.,} \quad m + y \cdot \delta = m - 1 + 2 \cdot \delta \tag{10.8}$$

$$\therefore \quad y = 2 - 1/\delta \tag{10.9}$$

and then $y = 1/\delta - 2 \cdot x = 2 - 1/\delta$ implying that $x = 1/\delta - 1$. This completes the analysis of $H(\delta, k)$ when k is a continuous variable. However, the function in (10) is not continuous, but discrete for integer values of k. The second difference then, is a sampling of the waveform in Fig. 1. This sampling yields one of the following sequences,

$$\ldots, 0, r, -r, 0, \ldots, 0, r, -r, 0, \ldots, 0, r, -r, 0, \ldots \qquad (11$$

when $\delta < 1/2$, or

$$\ldots, 0, -r, r, 0, \ldots, 0, -r, r, 0, \ldots, 0, -r, r, 0, \ldots \qquad (11l$$

when $\delta > 1/2$. When $\delta = 1/2$, there are no zeros in the sequence. Also the number zeros between any two $(r, -r)$ pairs can vary by 1.

To see how the sequences of (11) are obtained, consider the period of the wavefor in Fig. 1 and the values of x and y. Since $\delta < 1$, the period of the waveform, $1/\delta$, greater than 1. In the case where $\delta < 1/2$, $x = 1$ and it is easy to see how samplin the waveform for integer values of k produces the sequence of (11a).

To demonstrate that the sequence of (11b) is obtained when $\delta > 1/2$, it will shown that a sample value of 0 can never follow a sample value of $-r$. Also, it is n possible to sample two consecutive values of $-r$ or two of r.

Consider the values associated with the waveform in Fig. 1 as discussed abov Since, $1/2 < \delta < 1$, the period is bounded by $1 < 1/\delta < 2$. The value of x $1/\delta - 1$, and y is $2 - 1/\delta$. Recall that y is the length of the period where the functio is zero. Also, note that $x + y = 1$. This ensures that, if a value of r is sampled some point on the curve, the next point where the curve is equal to r is a distanc greater than $x + y$ away or a distance less than x away. Since $x < 1$, $x + y =$ and the curve is sampled at integer points, a sample of r may not immediately follow sample of r. The same is true for samples of $-r$. Similar reasoning also shows that sample of 0 may not follow a sample of $-r$. Because the sample interval is 1 an $x + y = 1$, a sample of $-r$ is always followed by a sample of r. However, a sample of or $-r$ may follow a sample of r depending on the relative values of x and y.

The approximate number of zeros between the $(r, -r)$ pairs in (11) can also determined. In the case where $\delta \leq 1/2$, the number of zeros is

$$L_0 = 1/\delta - 2 \quad . \qquad (12$$

This follows because $1/\delta > 2$ and L_0 is the number of samples in one period, less th two samples for r and $-r$.

For the $\delta > 1/2$ case, we show that $H(\delta, k) = -H(1 - \delta, k)$. First note that if is an integer and $\epsilon > 0$, $\lfloor j - \epsilon \rfloor = j - 1 - \lfloor \epsilon \rfloor$. Now let $\gamma = 1 - \delta$ implying tha $\delta = 1 - \gamma$. For the $\delta > 1/2$ case then, $\gamma < 1/2$. Substituting for δ in each of the thre terms in (10.1) yields

$$\alpha_{+1} = \lfloor (k+1) \cdot \delta \rfloor = \lfloor (k+1) \cdot (1-\gamma) \rfloor \qquad (12.$$

$$= \lfloor (k+1) - (k+1) \cdot \gamma \rfloor \qquad (12.$$

$$= k + 1 - 1 - \lfloor (k+1) \cdot \gamma \rfloor \quad , \qquad (12.$$

and

$$\alpha_0 = \lfloor k \cdot \delta \rfloor = \lfloor k \cdot (1-\gamma) \rfloor \qquad (12.4$$

$$= \lfloor k - k \cdot \gamma \rfloor \qquad (12.5$$

$$= k - 1 - \lfloor k \cdot \gamma \rfloor \quad , \qquad (12.6$$

and

$$\alpha_{-1} = \lfloor (k-1)\cdot\gamma \rfloor = \lfloor (k-1)\cdot(1-\gamma) \rfloor \tag{12.7}$$

$$= \lfloor (k-1)-(k-1)\cdot\gamma \rfloor \tag{12.8}$$

$$= k-1-1-\lfloor (k+1)\cdot\gamma \rfloor \quad . \tag{12.9}$$

Combining (12.3), (12.6) and (12.9) we get

$$H(\delta, k) = \alpha_{+1} - 2\alpha_0 + \alpha_{-1} \tag{12.10}$$

$$= k - \lfloor (k+1)\cdot\gamma \rfloor - 2\cdot k + 2 + 2\cdot\lfloor k\cdot\gamma \rfloor + k - 2 - \lfloor (k-1)\cdot\gamma \rfloor \tag{12.11}$$

$$= -\lfloor (k+1)\cdot\gamma \rfloor + 2\lfloor k\cdot\gamma \rfloor - \lfloor (k-1)\cdot\gamma \rfloor \tag{12.12}$$

$$= -H(1-\delta, k) \quad . \tag{12.13}$$

Thus, from (12a), the number of zeros in this case is simply

$$L_0 = \frac{1}{1-\delta} - 2 \quad . \tag{12b}$$

The above result also explains why the sequence of (11a) is the negative of the sequence of (11b).

L_0, then, can be controlled empirically by adding instructions to the loop calling the CLOCK function. This procedure adjusts the value of δ.

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

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

3.3. Operating System Interference

The isolation of the feature to be measured from other language features and code optimization is not the only isolation which must be achieved. The timing of the feature to be measured must also be isolated from times of other user processes or of 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 which 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 a question of how precisely this calculation is made. Therefore, benchmark tests should 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 because a request for output could create an independent process that runs concurrently with the benchmark.

Even with this disabling of daemon processes and running on a single user system, there are still timing anomalies to be detected and measured, most notably time sharing activities of the operating system. The operating system can still be expected to interrupt the benchmark periodically, check the queue for other processes waiting to run, and return control to the benchmark process. Also, for sufficiently high use of memory, operating system paging functions may be invoked. However, except for memory allocation/deallocation tests, benchmarks can usually be designed to use less memory than the size which 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. 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

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

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

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

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

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 $-1 < \epsilon_c, \epsilon_{cf} < 1$. And then

$$n_{cf} - n_c = \frac{1}{T_p} \cdot \left[N \cdot T_F + (n_{cf} - n_c) \cdot T_{os} \right] + \epsilon_{cf} - \epsilon_c \,. \tag{19.1}$$

Let $\beta = \frac{T_{os}}{T_p}$, and simplify to get

$$n_{cf} - n_c = \frac{N \cdot T_P}{T_p} + (n_{cf} - n_c) \cdot \beta + \epsilon_{cf} - \epsilon_c \quad . \tag{19.2}$$

Solving for $n_{cf} - n_c$ yields

$$(n_{cf} - n_c) = \frac{N \cdot T_P}{T_p \cdot (1-\beta)} + \frac{1}{1-\beta} \cdot (\epsilon_{cf} - \epsilon_c) \quad . \tag{19.3}$$

Combine with (17) to get

$$T_P = \frac{T_d}{N} - T_P \cdot \frac{T_{os}}{T_p \cdot (1-\beta)} - \frac{T_{os} \cdot (\epsilon_{cf} - \epsilon_c)}{N \cdot (1-\beta)} - (\delta_{cf} - \delta_c) \cdot \frac{\tau}{N} \tag{19.4}$$

which can be simplified to

$$\left[1 + \frac{\beta}{1-\beta}\right] \cdot T_P = \frac{T_d}{N} - \frac{T_{os} \cdot (\epsilon_{cf} - \epsilon_c)}{N \cdot (1-\beta)} - (\delta_{cf} - \delta_c) \cdot \frac{\tau}{N} \quad . \tag{19.5}$$

Since $1 + \frac{\beta}{1-\beta} = \frac{1}{1-\beta}$, multiply both sides by $1 - \beta$ to get

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

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

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

The two right most terms in (20) can be made arbitrary small by choosing N sufficiently large. The effect of β shows that the results previously obtained in (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 $T_{os} >> \tau$, and that $T_p = m \cdot \tau$ for $m >> 1$, and that δ in (5) is zero. The latter assumption means that the contribution to the second difference from the resolution time, τ, itself 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

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

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

If $\delta \neq 0$, then the above sequence will have the sequence of (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 $T_{os} >> \tau$, reasonable estimates of the parameters should still be obtainable. Theoretically, it is possible to derive the precise value of T_{os} based upon the number periods in

(21) between fluctuations of size τ in the values. In practice this will be difficult t detect because of the length of sequence required and the distortion from the $(\tau,-$ occurrence as in (11).

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

Minor extensions of this technique allow multiple periodic operating system func tions of differing service times to be detected and evaluated. However, it is generall very difficult to fit the execution time and period of more than a single function to th sequence of (11). Nevertheless, by accumulating the shortening of the strings of zero and dividing by total time, it is typically possible to get an overall estimate of th operating system overhead involved.

Actual tests with this approach revealed another difficulty. Some implementation of the CLOCK function involve the dynamic allocation of records, which in turn ma involve the invocation of a run-time system function. As will be discussed in Sec. 5.2 the time required to perform this operation can vary quite widely. This variation i storage allocation time will give the appearance of operating system overhead. To avoi these problems, the Ada CLOCK function should not be used for tests to determine th operating system overhead. Instead, an implementation dependent subprogram i required which can read the system timer without invoking any variable time syster functions such as storage allocation. Such a system dependent subprogram was writte and used in our tests. It should be noted, however, that for all of the other tests to b described, the CLOCK function is evaluated only at the beginning and the end of a loo iterated a large number of times, and the effect of the dynamic storage allocation i effectively eliminated, as shown in (20). Thus, except for determining the operating system overhead, the Ada CLOCK function may be safely used.

3.4. Resolution of Measurements

The result of (20) was based upon a periodically occurring function which alway took the same time to execute. In practice this assumption may not be entirely true Repeated executions of the benchmark can provide both a test of the validity of th assumptions and improve the accuracy of the results obtained.

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

An alternative strategy is to use the minimum of the values obtained. However, in this case, one must be careful 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 which is distorted on the side of being too small.

4. Features to be Measured

This section of the paper describes features which are relevant to real-time execution and whose performance should be measured. A motivation is given for each test as well as a precise statement of what is being measured. Where the measurement requires techniques beyond those described in Sec. 3, specific details are given.

The specific features discussed are:

- 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.

All but the last three are clearly measurements of features specified in the LRM.

In the areas of tasking, timing and storage management, the compiler implementors have been given a great deal of implementation latitude. Consequently, it is difficult to develop *a priori* a set of benchmarks which completely characterize these areas since the range of implementation techniques which may be used is open ended. Knowledge of the type of disciplines implemented is important before it can be determined what parameters it is relevant to measure. Thus, measurement techniques in these areas are oriented toward determining the general nature of the implementation techniques used.

4.1. Subprogram Overhead

With today's software systems running into sizes that exceed one million lines, modular programming is a necessity. Such a programming style, however, leads to an increase in procedure and function calls. In a recent study, Zeigler and Weicker found that 26.8% of a typical Ada program as implemented in the iMAX 432 system was subprogram calls [7]. Shimasaki, et al., obtained a range of 26.5% to 41.4% for typical Pascal systems [8]. The overhead associated with a subprogram call and return should not deter software producers from using a structured programming style. A possible way to avoid the cost of this increased overhead is to have 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 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 to different aspects of subprogram calls. The first test measures the raw overhead involved in entering and exiting a subprogram with no parameters. Next various numbers of INTEGER and ENUMERATION parameters are passed to determine the overhead associated with simple parameter passing. 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 case 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 of 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 overhead between intra- and inter-package calls. In the case of intra-package calls, all of 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 an 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 of the tests for both inter-package and intra-package calls are repeated with the subprograms being part of a generic unit. This test is designed to determine the additional overhead involved in executing generic instantiations of the code.

4.2. Dynamic Allocation of Objects

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

Three types of allocation are considered. The first case is that of allocating a fixed amount of storage by either entering a subprogram or a declare block with the objects declared locally. Although the amount of storage needed is known at compile time, it is allocated at run time. The second case is the allocation of a variable amount of storage not known at compile time by entering either a subprogram or declare block. An example of such an object would be an array with variable bounds. The third case of dynamic allocation is that done explicitly with the new allocator. This allocator can be used to allocate a single object of a particular type.

The tests presented measure the overhead associated with each type of dynamic allocation. 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 if there is any difference in the overhead based on the type of object allocated. In the variable length case, arrays of various dimensions bounded by variables are allocated. This test is designed to determine if allocation time is dependent on size of the object. In particular, it is expected that many compilers will allocate small objects on the stack assigned to the task, and larger objects off 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 are measured. This test will again show if allocation time is dependent on size (in the composite type object case). Also, these measurements will give an idea as to the relative efficiency of this method of allocation as opposed to the fixed length case.

4.3. Exceptions

Embedded real-time systems require extensive error-handing and recovery so that errors may 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 (space ships, satellites, etc.), the ability to provide extensive exception handling is of great importance. In order for these real-time systems to operate properly, efficient exception handling must be available.

Four types of exception handling routines are interesting since they represent different ways in which exceptions are raised: NUMERIC_ERROR, CONSTRAINT_ERROR, TASKING_ERROR, and user-defined exceptions. The NUMERIC_ERROR exception is first discovered by the hardware and the exception is 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 method of raising the exceptions can be done both by forcing the relevant abnormal state in the code and by using the **raise** statement.

In order to gauge the efficiency of exception handling, measurements of time to both respond to and propagate exceptions must be examined. The response time for an exception is the time between the raising of the exception and the start of the execution of the exception handler. 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 was 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. Where applicable, the exceptions in the tests are raised both by the raise statement and by forcing the abnormal state to occur in the code.

4.4. Task Elaboration, Activation, and Termination

The tasking function provides the heart of the real-time power and usefulness of Ada. Many algorithms, such as buffering algorithms, involve the creation and execution of tasks, e.g., the reader-writer scheme described in Barnes [9]. Nevertheless, task elaboration, activation and termination are almost always suspect operations in real-time programming and programmers often allocate tasks statically to avoid run-time execution time. It is, therefore, of special interest to explore the efficiencies of task elaboration and activation.

The time measured in this test is the time to elaborate a task's specification, activate the task, and terminate 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 provide more detailed information about the efficiency of tasking overhead. However, the coarse resolution of the CLOCK function currently available prevented measurement of the individual values, due to the large number of iterations needed to get a precise measurement. Iterating through a loop a large number of times 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.

Some additional information can be determined about the time for task activation, however. The test for measuring the composite of elaboration, activation, and termination is run for the two possible cases of task activation: 1) entering the non-declarative part of a parent block and 2) by using the **new** allocator. The first case can be further divided into two categories. The task to be activated can either be declared directly in the declarative part of a block, or it can be an object declared to be of a task type. In the case of task activation 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.

4.5. Task Synchronization

Important in multi-tasking is the ability of tasks to synchronize. In Ada, synchronization is supported in the rendezvous mechanism. This mechanism 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 it is ready to accept the rendezvous. The run-time system must check if the acceptor is indeed ready to receive the rendezvous and this adds to the overhead associated with the context switches. If the overhead associated with a rendezvous is too great, then the efficiency of execution in a multi-tasking environment will suffer.

The synchronization test measures the time to complete a rendezvous between a task and a procedure with no additional load present. This method, then, gives a lower bound on rendezvous time because no extraneous units of execution are competing for the CPU. This test is also repeated for rendezvous where various numbers, types and modes of parameters are passed.

4.6. 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 measures the overhead associated with a call to and a return from the CLOCK function provided in the package CALENDAR. The method used is essentially the same as the one used to measure the overhead associated with a entry and exit of a do-nothing subprogram in a separate package.

4.7. 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 of such a computation 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. This could be critical in the case of small delays.

The objective of the test in this case 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 just variables and both constants and variables. Although both "+" functions are essentially the same (only the order of parameters reversed), both are tested. This is done because 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.

4.8. Scheduling Considerations

Two requirements of many real-time programs are the need to schedule tasks to execute at particular points in time and the need to allow execution to switch among tasks. Ada provides the **delay** statement to allow programmers a mechanism for handling the former. The latter can be achieved through a variety of mechanisms. The scheduler provided by the run-time system is entered at certain synchronization points in a program, at which time other tasks may be placed into execution. Also, the underlying system may implement a time slice mechanism. Great freedom is provided Ada implementors in realizing these mechanisms, however, 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. These are the only points at which a user can be sure that the scheduler will be entered in a system which does not implement priorities. The issue that arises is determining when a task becomes eligible for execution after the expiration of a delay. An implementation may elect to only check for expiration of the delay periodically, at synchronization points, or in a variety of other ways.

To illustrate the problem consider an embedded system in which the programmer has control over all nonsystem tasks to be executed, and consider a simple polling loop whose purpose is to receive messages from a network device and post them to a local mailbox. While 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 form:

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

```
            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 given 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 to have a chance to run every time the message task runs, and that the message task would have a chance to run in accordance with underlying fair scheduling system. Further, if only the message task is ready to run, one would expect it to run approximately once every 50 milliseconds. However, if, as is the case in some validated compiler systems, the expiration of this delay is only checked periodically, say at 1 second 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 may be performed quite independently from time slicing or other task scheduling which 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. Even in this case, however, it is not clear when the implementation will check to determine if the delay has expired. This matter is presently under consideration of the Language Maintenance Committee, and it is thus wise to have a method for testing the scheduling algorithm used.

Even if fixed interval delay scheduling is used, acceptable performance may still be achieved under some circumstances if an implementation checks for tasks to schedule at points in addition to synchronization points. For example, if the loop given above is modified as shown below, other tasks may still obtain CPU service *if* the scheduler is entered to choose a new task to run each time a **select** is encountered.

```
loop
   select
      when DEVICE_HAS_MESSAGE =>
         accept RECEIVE(MESSAGE) do  -- MESSAGE is an out parameter
            DEPOSIT(MESSAGE);         -- procedure call
         end do;
      else
         null;
   end select;                                            (L3)
end loop;
```

Given a fair scheduler, some other task would then have an opportunity to execute each time around the loop. Of course, either the other tasks must relinquish control sufficiently often or the scheduler must time share with sufficient frequency so that the polling loop can regain control often enough. The price for the use of additional scheduling points is extra scheduling overhead.

In order to develop many real-time Ada programs it is thus clearly necessary to have supplemental information about the scheduling strategies used by an implementation. A method for determining the time slice interval was described earlier. In the next subsection, techniques for determining the scheduling discipline related to delay expiration are described.

4.8.1. Delay and Scheduling Measurements

This section proposes a test which allows information regarding preemptive or fixed interval scheduling to be obtained. The test is based upon embedding a simple **delay** statement inside of a loop executed a large number of times, for example:

```
T1 : = CLOCK;
while I < N loop                                (L4)
     delay DEL;
     INCREMENT(I);
end loop;
T2 : = CLOCK;
```

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

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

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 one, as shown in Fig. 2. The deviation of the plot from this ideal provides useful information about the scheduler.

4.8.1.1. Minimum Delay Overhead

First, it is necessary 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 a return to the calling unit immediately. 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, i.e., it is the overhead associated with delays. Typically, T_δ will depend upon 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.

Beginning with DEL = DURATION'SMALL make a series of runs of (L4).for increasing values of DEL, and generate the plot described above. Suppose d(DEL) remains constant for small values of DEL as shown in Fig. 3. This suggests that for DEL less than some components of T_δ, the system does an immediate return to the calling program (or immediate 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 the values of 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 nonpreemptive.

4.8.1.2. Fixed Interval vs Preemptive Delay Scheduling

Next, we try to determine if fixed interval delay scheduling or true preemptive scheduling based upon interrupts from a programmable clock are 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 one is not achieved, it is suggestive that true preemptive scheduling is not being used. The plot is then likely to be a staircase function if fixed interval delay scheduling is being used. To see this consider that only this task 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 Fig. 4. Again, obtaining a sufficient set of values for d(DEL) is not entirely straightforward. Some compilers are known to have a scheduling interval more than five orders of magnitude larger than DURATION'SMALL. Therefore, some cleverness is required in selecting the values of DEL to use, e.g., a coarse to fine search strategy.

There is one additional characteristic to a scheduling strategy which might complicate the interpretation somewhat. 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, r, is relatively small compared to $T1-T2$ for $N=1$, the two cases 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 while for fixed interval delay scheduling, $T2-T1$ will vary randomly with the range of variation corresponding to the size of the interval of the scheduler.

4.8.1.3. Compensation for Minimum Delay Overhead

Finally, if preemptive scheduling has been used and DURATION'SMALL is significantly less than T_δ, further information can be obtained. 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 is indicative of either no compensation for T_δ or incorrect compensation. More generally, due to the dependence of T_δ upon DEL, the plot might be composed of several line segments, and one could examine each line segment 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.

While the data obtained in the test described above must be analyzed in several different ways, this test does provide information which allows a great deal of useful information to be determined about an implementation.

4.9. Memory Deallocation and Garbage Collection

Memory allocation and dellocation processes are often critical to the operation of real time systems. Systems can fail because there is insufficient (virtual) memory available, because the allocation or deallocation times are too large, or because a deallocation process (garbage collector) is implicitly called at times not under control of the applications program. (The authors are painfully aware of the latter possibility through personal experience.)

There are two reasons why insufficient memory failures might occur. First, there might just intrinsically be too little space available in the pool of storage from which allocations are made. For most systems, this problem will probably not occur. More importantly for real-time systems, however, is the fact that the LRM does not require an immediate return to the storage pool of the deallocated storage, and a validated compiler has been found which 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 while 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 storage allocation responsibility. Further, storage deallocation for real-time systems should 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 level. Garbage collection can then take a substantial length 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. For example, imagine a tight 1 millisecond control loop on an aircraft suddenly put into abeyance for a couple of seconds.

There are also interesting run-time system or operating system effects which one might wish to observe. For 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. For example, it has been found that in a Unix system, when the allocation storage approaches the virtual storage limit, overhead times of several seconds occur. (This is probably not a problem, however, since the virtual size limit is so large that rarely, if ever, would one run into this problem.)

The basic idea in building tests to measure the effects mentioned above is to use the new allocator in a loop with various controls on whether it is or is not possible for deallocation to take place. The second differencing techniques described in Sec. 3.2 can be used to measure the relevant times which occur.

For one test, a large array of pointers to a sizable array of data is declared. Then each time through the loop, a pointer to a newly 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
  for I in 1..MAX loop
    PTR_ARRAY(I) := new INT_ARRAY;
    TIME_ARRAY(I) := CLOCK;                          (L5)
  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 the storage allocation and paging times.

A second test uses the same loop structure, but only two access variables. Each time around the loop, the content of one access variable is shifted to the second, and the newly acquired data is assigned to the first access variable, thus implicitly freeing the storage allocated two iterations previous to the current one. (This shifting structure is used to break up the possibility of an optimizer avoiding 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 which can be allocated which is independent of the availability of storage to be allocated (a strange and unlikely occurrence).

These tests provide basic information on the storage allocation and deallocation mechanisms used by an Ada system.

4.10. Interrupt Response Time

Interrupt response time is clearly critical for many real-time embedded systems. Techniques for measuring it, however, are difficult to develop since, in general, hardware external to the CPU must be involved, i.e., the test cannot be based only on programming. Second, the times which must be measured will be at substantively different points in the test program and the use of iteration to improve accuracy of measurement, as shown in (4), can not be expected to work in this situation.

The first problem to be faced is the generation of the interrupt signal in a controlled and time measurable fashion. This can be accomplished by adding a parallel interface to the system to be tested and writing a special driver for the interface which must be directly accessible from the benchmark program. The output from the parallel interface is treated as a logic signal to cause an interrupt to the processor. The procedure for outputting a signal through this interface must be written to be directly callable, and hence time measurable, from the benchmark program. The procedure must not have to go through the underlying run-time or operating system. Then, using techniques described earlier it will be 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 which 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                                    (L6)
    TIME_ARRAY(I) := CLOCK;
    SEND_SIGNAL;            -- to parallel interface & create interrupt
  end loop;
```

The second program segment is an interrupt handler which 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. Let T_{ov} be this time. Next calculate the average time difference between the times recorded in the main loop and the corresponding times recorded in the interrupt handler. Denote this average by T_{ave}. Then, one can calculate the interrupt response time as $T_{ave} - T_{ov}$.

5. Results

In this section we illustrate the application of the benchmarks by 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. All user and daemon processes were disabled (except for the swapper and page daemon, which can never be disabled). The tests described in Sec. 3 were run to determine the operating system overhead injected into the measurements for Unix on the VAX. The components to overhead individually required significantly less than the resolution of the time measurement, r. Thus, as indicated in Sec. 3.3, it was difficult to get an accurate value for the overhead. Nevertheless, by examining the amount by which the string of zeros is shortened we were able to obtain a crude estimate of the overhead. With this approach, we estimated the overhead to be 5%. Due 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 theoretically accurate to the nearest tenth of a microsecond or tenth 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 within 0.1 or 0.2 microseconds (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 absolutely 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 (related 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 just to be sure that our benchmarks were correct. We will describe some of these below as illustrations of the differences which can occur.

A consequence of such minor variations is that it is difficult to place meaning on results any closer than a couple of microseconds, even though theoretically more accurate results have been obtained. There are two reasons for this. First, the number of special cases to track down is sufficiently large as to require a very large effort to be complete. Second, even if one did track down each situation completely, there would be so many separate cases to report that one could not reasonably try to use all of the data anyway.

A summary of the test results is presented in the tables in Appendix A. Highlights are discussed here; a complete list is given in Appendicies B thru G.

5.1. Subprogram Overhead

A summary of the results of making procedure calls of various kinds is given in Table A.1. There are several surprises in this table. 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, non-generic, 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 non-generic versions of the code.

A second characteristic, not obvious from the table, is the effect of code optimization. The DEC compiler, in particular, will in-line procedures for small procedure sizes automatically as a time optimization even if INLINE is not used. While this improves performance substantially, it makes it difficult to test procedure calling time, and raises a question of interpretation of the results. The numbers not available for procedure calls in Table A.1 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 due to size of 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 was that the DEC peep-hole 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, while 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 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 upon 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 microseconds per argument.

5.2. Dynamic Allocation of Objects

The memory allocation tests, shown in Table A.2, 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. Thus, this 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 had been expected (considering the formulas typically used for computing array dope vectors). The times were in the 10-20 microsecond per dimension range for all compilers except the Verdix 5.1 and Alsys compilers, which were appreciably larger. All of the ranges used in these tests were kept small in order to avoid other storage effects, such as 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. This version never deallocated storage. Thus, 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. This paging time was sufficient to distort the test results. To eliminate this difficulty, a sequence of pretests were 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 the precision somewhat, but useful results were still obtained. Versions 5.1 and higher did deallocate storage, which, while they eliminated the paging problem, did increase slightly the storage times recorded.

Second, most of the compilers used a multi-level storage allocation scheme. Small objects were allocated from some locally held storage pool, while for larger objects, calls were made to the underlying system for more storage. The latter were quite evident since they typically required near 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 some object sizes from 4 to 4,000 bytes. The wide range of times shown in table A.2 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 (though it was not easy to detect) in the CLOCK function. The Verdix CLOCK function dynamically allocates a record each time it is called. The time to allocate this record from the local pool is only a few 10's of microseconds. However, every once in a while the local pool becomes exhausted, and a system call must be made to obtain more storage. The time to obtain a new chunk of storage is on the order of 3 milliseconds. 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 real-time programs, as CLOCK will be used in many, if not most, real-time scheduling loops. Since the system call for more storage doesn't happen very often, it will be difficult to isolate the problem. Consequently, it is important to identify all implementation supplied procedures or functions which allocate storage.

5.3. 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 which 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, violation of a subtype range, and INTEGER overflow.

The results of the exceptions tests are shown in Table A.3. In general, the compilers all take little or no time for exceptions that are not raised, which is an important characteristic for real-time applications. However, all of the exception handing 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 program rather than relying upon 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 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 test results on a time-shared system.

5.4. Task Elaboration, Activation, and Termination

This test was run for the three different types of task elaboration and activation explained in Sec. 4.4. Table A.4 shows the task elaboration, activation and termination times for the compilers tested. For each individual compiler the differences between elaboration and activation in a declarative region or via the **new** operator did not differ by more than 15% and thus are not reported separately. The table shows that efficient techniques for task elaboration, activation, and termination are possible.

5.5. Task Synchronization

The test here was rather straightforward. The test involved entering a block where a task was activated and a subprogram called that executed 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. The results are also shown in Table A.4. 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 to pass parameters was negligible.

5.6. Clock Function Overhead and Resolution

Table A.5 shows the overhead associated with the CLOCK function. The numbers reported are averages obtained over several test runs. There is 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 savings 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. While useful for some applications, these extensive computations are too time-expensive for many real-time applications, and some additional clock like function will be required for real-time applications. A CLOCK resolution of 10 milliseconds is marginal for many real-time applications.

5.7. Arithmetic for types TIME and DURATION

The TIME math tests measure overhead involved in addition and subtraction operations involving the types TIME and DURATION. All possible combinations involving variables and constants of each type are tested. Table A.6 shows the results. It appears that constant expressions are evaluated at compile time in all of the compilers. The difference of more than an order of magnitude between operations on the type TIME and the type DURATION is probably due to the representation of TIME as a record, while DURATION is fixed point. The variation in the results between the versions of the Verdix compiler for expressions involving type TIME is due to a change in the record used to represent TIME.

5.8. Delay and Scheduling Measurements

This test involved the measurement of time elapsed during the execution of a **delay** statement. The results appear in Table A.5.

For the Verdix 4.06 compiler, a minimum delay value of 1.4 milliseconds was detected. This delay occurred for requested delays between zero and slightly less than 1 millisecond (actually, the upper bound is 16 times DURATION'SMALL, the greatest model number less than 1 millisecond). This value corresponds to the part of the curve before the jump in Fig. 4. For the Verdix 5.2 and DEC compilers the minimum actual delay was 10 milliseconds, and while for the Alsys compiler it was 1 second.

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 second. Thus, for a requested delay of 1 millisecond or greater, the actual delay was for the remainder of the one second 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 test confirmed that requested delays between 1 millisecond and less than 10 milliseconds resulted in actual delays between 10 milliseconds and 1.01 seconds, that is, the value remaining in the 1 second time slice plus 10 milliseconds. As the requested delay was increased, the staircase function of Fig. 4 was obtained, with the step size being 1.01 seconds. The extra 0.01 seconds corresponds to one clock resolution time and appears to be time spent in the scheduler before the basic 1 second time slice is reset.

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

5.9. Storage Deallocation and Garbage Collection

The storage deallocation tests provided an insight to 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 megabytes, the limit imposed by the operating system. This is the size at which STORAGE_ERROR should be raised by (L5).

By modifying the test loop to use only two access variables, instead of the array of access variables in (L5), we found that the 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.

The 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. Due to improper setting of system parameters on the Microvax II, the DEC VAX 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.

6. Summary and Conclusion

This paper has developed 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 the test results. In order 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 in which the tests are being run, a simple test of repeatedly reading the system clock and analyzing the results to identify the frequency and size of operating system activity should be performed before running the tests.

Although the benchmarks are intended for testing real-time performance, the only Ada systems available to us at the time of development were intended for time-shared, and not real-time, use. Time shared systems often place less emphasis on the real-time performance than on general program development and execution support, and the results of applying our tests bore this out. However, by the same token, the results point to areas in which users should expect significant performance improvements in systems intended for real-time applications. Among the areas so noted are: 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 also a small number of real-time relevant tests which we were not able to perform on the systems available to us, i.e., the interrupt response time and the behavior of the system with respect to task scheduling upon I/O requests. A test was proposed for the former, and the time-shared operating system determines the behavior of the latter 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, we make several observations based upon our experience in developing these benchmarks. First, so many implementation dependent variations are validateable that it is not safe, in our opinion, to use an Ada compiler for real-time applications without first checking it with performance evaluation tools. Characteristics such as time management, scheduling and memory mangement can have validated implementations that will devastate a real-time application. 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.

7. Acknowledgements

The authors wish to thank Chuck Antonelli for sharing his knowledge of the Unix operating system and his help in obtaining and interpreting direct time readings from Unix, and Ron Theriault, Jarir Chaar, and Sue Hsieh for numerous late nights in helping run the benchmark tests. We are also grateful to 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. We are also appreciative of assistance from Rob Bedichek and Steve Ziegler of Verdix.

8. References

[1] *Ada programming language (ANSI/MIL-STD-1815A).* Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD(R&D), Jan. 1983.

[2] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Jour.*, vol. 19, no. 1, pp. 43-49, Feb. 1973.

[3] R.P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM,* vol. 27, no. 10, pp. 1013-1030, October 1984.

[4] P.J. Jalics, "Comparative performance of cobol vs pl/1 programs," *Computer Performance Evaluation Users Group 16th Meeting,* Oct. 20-23, 1980.

[5] J. Squire, "Performance issues workshop," *ACM SIGADA Users Committee Performance Issues Working Group,* July 15-16, 1985.

[6] M.J. Bassman, G.A. Fisher,Jr., and A. Gargaro, "An approach for evaluating the performance efficiency of Ada compliers," *Ada in Use, Proc. of the Ada Int'l Conf.*, May 14-16, 1985.

[7] S.F. Zeigler and R.P. Weiker, "Ada language statistics for the iMAX 432 operating system," *Ada Letters*, vol. 2, no. 6, pp. 63-67, May 1983.

[8] M. Shimasaki, S. Fukaya, K. Ikeda, and T. Kiyono, "An analysis of pascal programs in compiler writing," *Software Practice and Experience*, vol. 10, no. 2, pp. 149-157, Feb. 1980.

[9] J.G.P. Barnes, *Programming in Ada.* London: Addison-Wesley Publishing Co., 1984.

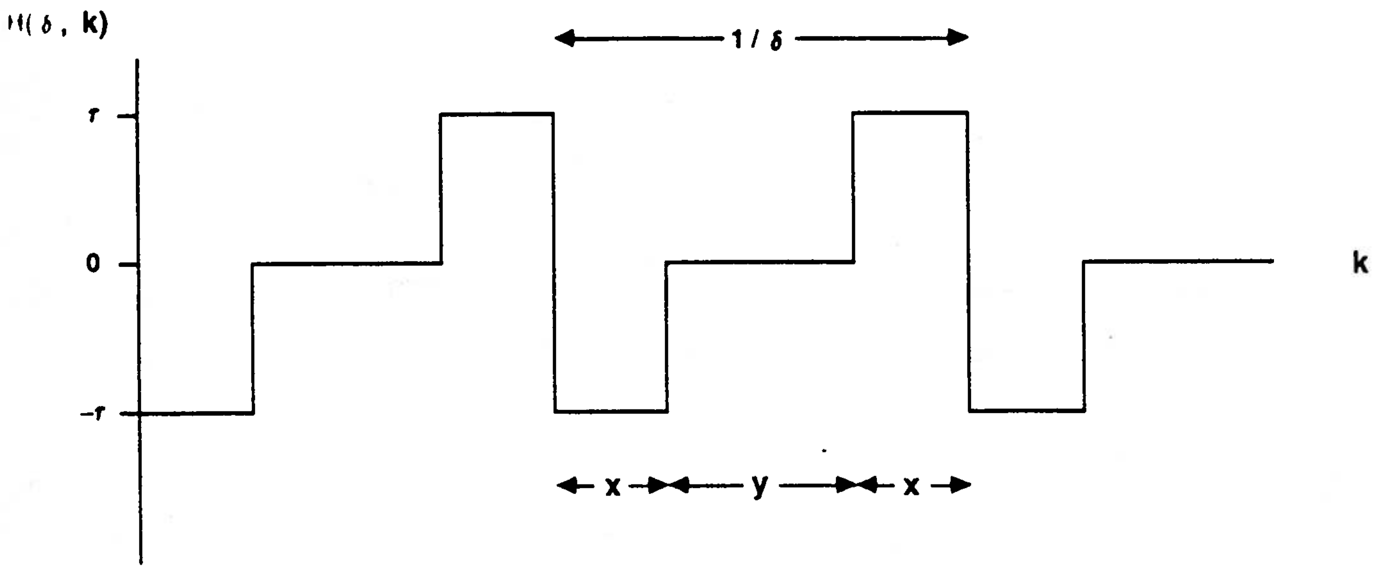

Figure 1. H(δ, k) v. k

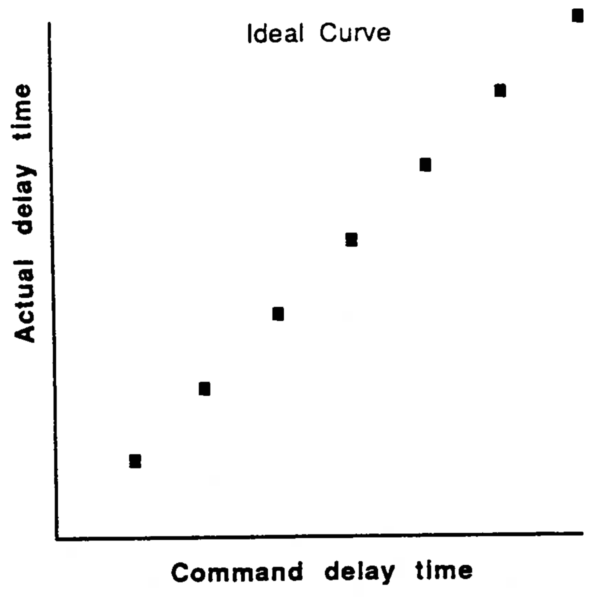

Figure 2. The Ideal Delay Curve

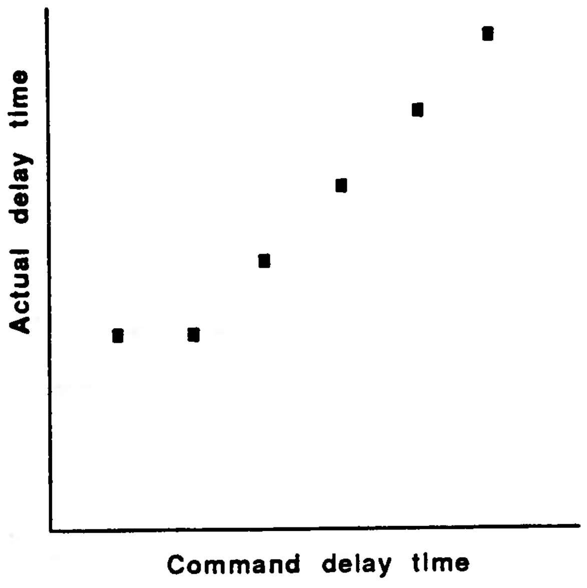

Figure 3. The Delay Curve for Small Values of DEL Showing Minimum Delay Overhead.

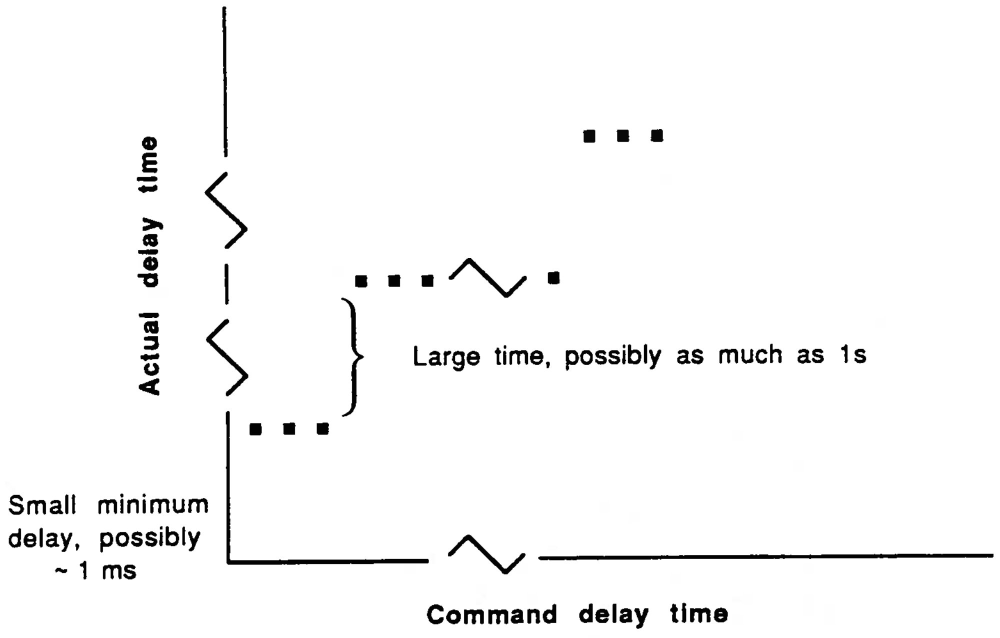

Figure 4. The Delay Curve for Fixed-Interval Scheduling

APPENDIX A

Benchmark Results

The following tables summarize the results of the benchmark tests run. All values are reported in microseconds except where noted. The compiler/hardware combinations tested are:

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 A.1

Procedure Calls, no arguments

Compiler	Inter-package	Intra-package	Inline	Generic Interpackage	Generic Intrapackage
Verdix 4.06	17.7μsec	27.7μsec	28.6μsec	25.9μsec	30.5μsec
Verdix 5.1	17.6	16.8	0.4	18.5	17.6
Verdix 5.2	18	17	0.0	18	20
Alsys 1.0	14	12	31	8	27
DEC VAX Ada, V.1.1	46.1	x[1]	3.0	45.9	x[1]
DEC VAX Ada, V.1.3	27.0	x[1]	0	15	x[1]

Additional overhead per integer argument

Compiler	in	out	in out
Verdix 4.06	~1.5μsec	~3.0μsec	~3.0μsec
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+	~6+
DEC VAX Ada, V.1.3	~1.5	~3+	~6+

[1]Compiler INLINED the call reducing value to zero. DEC has supplied a value of 15.1μsec for this call.

Table A.2

Dynamic Storage Allocation

Dynamically bounded arrays[1] in Declarative Region

Compiler	1-D Array	2-D Array	3-D Array
Verdix 4.06	31μsec	46μsec	54μsec
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

Dynamically bounded arrays[2] allocated via *new*

Compiler	1-D Array	2-D Array	3-D Array
Verdix 4.06	221 - 284μsec	309 - 1200μsec	326 - x[3]μsec
Verdix 5.1	200 - 260	280 - 1140	300 - 3,370
Verdix 5.2	220 - 280	290 - 1300	300 - 3,350
Alsys 1.0	2,249 - 2,185	2,191 - 2,217	2,300 - 2,334
DEC VAX Ada, V.1.1	410 - 450	430 - 870	490 - 4,830
DEC VAX Ada, V.1.3	290 - 300	280 - 300	370

Compiler	**Fixed length objects (small, no arrays) located via *n***
Verdix 4.06	133 - 300μsec
Verdix 5.1	227 - 270
Verdix 5.2	130 - 239
Alsys 1.0	1,963 - 1,985
DEC VAX Ada, V.1.1	310 - 510
DEC VAX Ada, V.1.3	250

[1]Integer arrays with range 1 along each dimension.

[2]Two tests each, integer arrays with ranges 1 and 10 along each dimension.

[3]Storage errors resulted when we attempted to allocate larger amounts of storage.

Table A.3
Exception Handling

Compiler	User Defined, not Raised		Numeric Error, implicitly raised in Procedure	Other Exceptions	
	Block	Procedure		Handled in Block	Propagated to Calling Procedu
Verdix 4.06	0μsec	0μsec	2.47ms	345 - 402μsec	614 - 671μsec
Verdix 5.1	0	1	2.55	315 - 372μsec	544 - 613μsec
Verdix 5.2	0	1	2.72	396 - 448μsec	718 - 783μsec
Alsys 1.0	0	0	17.95	8.8 - 9.8ms	19 - 20ms
DEC VAX Ada, V.1.1	4	16	0.89	667 - 836μsec	736 - 894μsec
DEC VAX Ada, V.1.3	3	12	0.60	414 - 541μsec	482 - 619μsec

Table A.4
Tasking Times

Compiler	Rendezvous	Task Elaborate, Activate, Terminate
Verdix 4.06	3.50ms	19.6ms
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	1.1	6.6

Table A.5
Timing & Scheduling

Compiler	Clock Call	Clock Resolution	Delay Scheduling Method	Effective Delay Resolution
Verdix 4.06*	570μsec	10ms	fixed interval	Variable 10ms - 1sec
Verdix 5.1	3,550	10ms	fixed interval	Variable 10ms - 1sec
Verdix 5.2	3,644	10ms	preemptive	10ms
Alsys 1.0	1,500	1sec	?	1sec
DEC VAX Ada V.1.1	95	10ms	preemptive	10ms
DEC VAX Ada. V.1.3	89	10ms	preemptive	10ms

Table A.6
Time

Compiler	TIMEs only	DURATIONs only	DURATION := TIME - TIME;
Verdix 4.06	188 - 241μsec	7.2 - 7.6μsec	111μsec
Verdix 5.1	716 - 812	6.3	50
Verdix 5.2	816 - 889	6.0	75
Alsys 1.0	88 - 105	1 - 2	189
DEC VAX Ada, V.1.1	98 - 109	x[1]	118
DEC VAX Ada, V.1.3	91 - 94	1	94

[1]Reliable data unavailable.

APPENDIX B

The following pages contain result tables for all of the tests run. These results are for the Verdix Compiler Version 4.06 running with Unix 4.2 bsd on a Vax 11/780. Some values contain explainatory footnotes.

```
Compiler Time Related Values:
---------------------------------------------
System Tick=        0.009948730468750  Seconds
Duration Small=     0.000061035156250  Seconds
---------------------------------------------
```

Subprogram Overhead (non-generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
27.7		0		
30.3	I	1	INTEGER	
31.1	O	1	INTEGER	
31.9	I_O	1	INTEGER	
43.1	I	10	INTEGER	
58.1	O	10	INTEGER	
58.7	I_O	10	INTEGER	
182.1	I	100	INTEGER	
330.8(1)	O	100	INTEGER	
445.7(2)	I_O	100	INTEGER	
29.8	I	1	ENUMERATION	
31.0	O	1	ENUMERATION	
31.9	I_O	1	ENUMERATION	
43.7	I	10	ENUMERATION	
58.2	O	10	ENUMERATION	
58.1	I_O	10	ENUMERATION	
182.8	I	100	ENUMERATION	
353.7(3)	O	100	ENUMERATION	
601.4(4)	I_O	100	ENUMERATION	
30.3	I	1	ARRAY of INTEGER	1
33.0	O	1	ARRAY of INTEGER	1
33.0	I_O	1	ARRAY of INTEGER	1
52.6(5)	I	1	ARRAY of INTEGER	10
31.0	O	1	ARRAY of INTEGER	10
31.2	I_O	1	ARRAY of INTEGER	10
30.5	I	1	ARRAY of INTEGER	100
30.6	O	1	ARRAY of INTEGER	100
30.7	I_O	1	ARRAY of INTEGER	100
31.6	I	1	ARRAY of INTEGER	10000
31.2	O	1	ARRAY of INTEGER	10000
31.4	I_O	1	ARRAY of INTEGER	10000
30.2	I	1	RECORD of INTEGER	1
31.7	O	1	RECORD of INTEGER	1
31.9	I_O	1	RECORD of INTEGER	1
31.1	I	1	RECORD of INTEGER	100
31.1	O	1	RECORD of INTEGER	100
31.1	I_O	1	RECORD of INTEGER	100
32.2	I	1	UNCONSTRAINED ARRAY	1
32.2	O	1	UNCONSTRAINED ARRAY	1
32.1	I_O	1	UNCONSTRAINED ARRAY	1
32.6	I	1	UNCONSTRAINED ARRAY	100
32.4	O	1	UNCONSTRAINED ARRAY	100
32.4	I_O	1	UNCONSTRAINED ARRAY	100
32.3	I	1	UNCONSTRAINED ARRAY	10000
32.3	O	1	UNCONSTRAINED ARRAY	10000
32.2	I_O	1	UNCONSTRAINED ARRAY	10000
30.5	I	1	UNCONSTRAINED RECORD	1
31.8	O	1	UNCONSTRAINED RECORD	1
31.0(6)	I_O	1	UNCONSTRAINED RECORD	1
30.2	I	1	UNCONSTRAINED RECORD	100
31.8	O	1	UNCONSTRAINED RECORD	100
30.9(7)	I_O	1	UNCONSTRAINED RECORD	100

(1) - Results for this test have ranged from 330 to 390 microseconds.
(2) - Results for this test have ranged from 336 to 665 microseconds.
(3) - Results for this test have ranged from 340 to 361 microseconds.
(4) - Results for this test have ranged from 352 to 601 microseconds.
(5) - Other runs have indicated that this value is 30.8 microseconds.
(6) - Other runs have indicated that this value is 31.7 microseconds.
(7) - Other runs have indicated that this value is 31.8 microseconds.

Subprogram Overhead (inline)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
28.6		0		
29.9	I	1	INTEGER	
30.9	O	1	INTEGER	
31.7	I_O	1	INTEGER	
43.3	I	10	INTEGER	
63.4(1)	O	10	INTEGER	
58.4	I_O	10	INTEGER	
185.1(2)	I	100	INTEGER	
330.7(3)	O	100	INTEGER	
363.2(4)	I_O	100	INTEGER	
29.9	I	1	ENUMERATION	
31.0	O	1	ENUMERATION	
32.1	I_O	1	ENUMERATION	
42.7	I	10	ENUMERATION	
58.3	O	10	ENUMERATION	
58.1	I_O	10	ENUMERATION	
182.1	I	100	ENUMERATION	
335.9(5)	O	100	ENUMERATION	
347.0(6)	I_O	100	ENUMERATION	
30.2	I	1	ARRAY of INTEGER	1
32.5	O	1	ARRAY of INTEGER	1
32.7	I_O	1	ARRAY of INTEGER	1
30.8	I	1	ARRAY of INTEGER	10
30.8	O	1	ARRAY of INTEGER	10
31.3	I_O	1	ARRAY of INTEGER	10
30.4	I	1	ARRAY of INTEGER	100
30.0	O	1	ARRAY of INTEGER	100
30.5	I_O	1	ARRAY of INTEGER	100
31.4	I	1	ARRAY of INTEGER	10000
31.3	O	1	ARRAY of INTEGER	10000
31.3	I_O	1	ARRAY of INTEGER	10000
30.5	I	1	RECORD of INTEGER	1
32.4	O	1	RECORD of INTEGER	1
34.1(7)	I_O	1	RECORD of INTEGER	1
31.0	I	1	RECORD of INTEGER	100
31.3	O	1	RECORD of INTEGER	100
31.1	I_O	1	RECORD of INTEGER	100
32.1	I	1	UNCONSTRAINED ARRAY	1
32.3	O	1	UNCONSTRAINED ARRAY	1
32.2	I_O	1	UNCONSTRAINED ARRAY	1
32.2	I	1	UNCONSTRAINED ARRAY	100
32.3	O	1	UNCONSTRAINED ARRAY	100
32.3	I_O	1	UNCONSTRAINED ARRAY	100
32.3	I	1	UNCONSTRAINED ARRAY	10000
32.2	O	1	UNCONSTRAINED ARRAY	10000
32.2	I_O	1	UNCONSTRAINED ARRAY	10000
30.5	I	1	UNCONSTRAINED RECORD	1
31.7	O	1	UNCONSTRAINED RECORD	1
31.7	I_O	1	UNCONSTRAINED RECORD	1
30.5	I	1	UNCONSTRAINED RECORD	100
31.7	O	1	UNCONSTRAINED RECORD	100
31.7	I_O	1	UNCONSTRAINED RECORD	100

(1) - Other runs have indicated that this value is 58.1 microseconds.
(2) - Results for this test have ranged from 185 to 242 microseconds.
(3) - Results for this test have ranged from 330 to 348 microseconds.
(4) - Results for this test have ranged from 363 to 378 microseconds.
(5) - Results for this test have ranged from 335 to 375 microseconds.
(6) - Results for this test have ranged from 342 to 405 microseconds.
(7) - Other runs have indicated that this value is 32.0 microseconds.

Subprogram Overhead (generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
30.5		0		
34.6	I	1	INTEGER	
35.7	O	1	INTEGER	
35.8	I_O	1	INTEGER	
48.2	I	10	INTEGER	
63.3	O	10	INTEGER	
63.5	I_O	10	INTEGER	
165.3(1)	I	100	INTEGER	
355.3(2)	O	100	INTEGER	
393.5(3)	I_O	100	INTEGER	
34.3	I	1	ENUMERATION	
35.9	O	1	ENUMERATION	
35.8	I_O	1	ENUMERATION	
48.0	I	10	ENUMERATION	
62.8	O	10	ENUMERATION	
63.4	I_O	10	ENUMERATION	
199.9(4)	I	100	ENUMERATION	
369.3(5)	O	100	ENUMERATION	
350.9(6)	I_O	100	ENUMERATION	
35.0	I	1	ARRAY of INTEGER	1
36.6	O	1	ARRAY of INTEGER	1
36.3	I_O	1	ARRAY of INTEGER	1
35.7	I	1	ARRAY of INTEGER	10
35.4	O	1	ARRAY of INTEGER	10
35.8	I_O	1	ARRAY of INTEGER	10
35.6	I	1	ARRAY of INTEGER	100
35.6	O	1	ARRAY of INTEGER	100
35.6	I_O	1	ARRAY of INTEGER	100
35.3	I	1	ARRAY of INTEGER	10000
35.3	O	1	ARRAY of INTEGER	10000
35.4	I_O	1	ARRAY of INTEGER	10000
35.0	I	1	RECORD of INTEGER	1
36.5	O	1	RECORD of INTEGER	1
36.5	I_O	1	RECORD of INTEGER	1
35.4	I	1	RECORD of INTEGER	100
35.3	O	1	RECORD of INTEGER	100
35.4	I_O	1	RECORD of INTEGER	100

(1) - Results for this test have ranged from 165 to 190 microseconds.
(2) - Results for this test have ranged from 355 to 397 microseconds.
(3) - Results for this test have ranged from 344 to 393 microseconds.
(4) - Results for this test have ranged from 186 to 200 microseconds.
(5) - Results for this test have ranged from 321 to 550 microseconds.
(6) - Results for this test have ranged from 350 to 471 microseconds.

Subproghram Overhead (cross package, non-generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
17.7		0		
19.4	I	1	INTEGER	
21.0	O	1	INTEGER	
21.2	I_O	1	INTEGER	
31.6	I	10	INTEGER	
46.6	O	10	INTEGER	
46.7	I_O	10	INTEGER	
196.4(1)	I	100	INTEGER	
323.6(2)	O	100	INTEGER	
324.5(3)	I_O	100	INTEGER	
19.6	I	1	ENUMERATION	
20.6	O	1	ENUMERATION	
20.6	I_O	1	ENUMERATION	
31.5	I	10	ENUMERATION	
50.9(4)	O	10	ENUMERATION	
46.6	I_O	10	ENUMERATION	
170.1(5)	I	100	ENUMERATION	
322.4(6)	O	100	ENUMERATION	
335.6(7)	I_O	100	ENUMERATION	
19.9	I	1	ARRAY of INTEGER	1
21.1	O	1	ARRAY of INTEGER	1
20.4	I_O	1	ARRAY of INTEGER	1
19.6	I	1	ARRAY of INTEGER	10
19.6	O	1	ARRAY of INTEGER	10
19.6	I_O	1	ARRAY of INTEGER	10
19.2	I	1	ARRAY of INTEGER	100
19.6	O	1	ARRAY of INTEGER	100
19.6	I_O	1	ARRAY of INTEGER	100
19.6	I	1	ARRAY of INTEGER	10000
19.1	O	1	ARRAY of INTEGER	10000
19.1	I_O	1	ARRAY of INTEGER	10000
20.6	I	1	RECORD of INTEGER	1
21.2	O	1	RECORD of INTEGER	1
21.1	I_O	1	RECORD of INTEGER	1
19.8	I	1	RECORD of INTEGER	100
19.6	O	1	RECORD of INTEGER	100
19.6	I_O	1	RECORD of INTEGER	100
23.2	I	1	UNCONSTRAINED ARRAY	1
23.4	O	1	UNCONSTRAINED ARRAY	1
23.3	I_O	1	UNCONSTRAINED ARRAY	1
23.3	I	1	UNCONSTRAINED ARRAY	100
23.3	O	1	UNCONSTRAINED ARRAY	100
23.3	I_O	1	UNCONSTRAINED ARRAY	100
23.2	I	1	UNCONSTRAINED ARRAY	10000
23.3	O	1	UNCONSTRAINED ARRAY	10000
23.3	I_O	1	UNCONSTRAINED ARRAY	10000
20.5	I	1	UNCONSTRAINED RECORD	1
22.4	O	1	UNCONSTRAINED RECORD	1
22.4	I_O	1	UNCONSTRAINED RECORD	1
20.4	I	1	UNCONSTRAINED RECORD	100
22.2	O	1	UNCONSTRAINED RECORD	100
22.3	I_O	1	UNCONSTRAINED RECORD	100

(1) - Results for this test have ranged from 170 to 196 microseconds.
(2) - Results for this test have ranged from 323 to 347 microseconds.
(3) - Results for this test have ranged from 323 to 380 microseconds.
(4) - Other runs have indicated that this value is 46.6 microseconds.
(5) - Results for this test have ranged from 170 to 182 microseconds.
(6) - Results for this test have ranged from 322 to 391 microseconds.
(7) - Results for this test have ranged from 331 to 360 microseconds.

Subprogram Overhead (generic, cross package)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
25.9		0		
30.9	I	1	INTEGER	
32.3	O	1	INTEGER	
32.6	I_O	1	INTEGER	
43.8	I	10	INTEGER	
57.6	O	10	INTEGER	
58.3	I_O	10	INTEGER	
185.2	I	100	INTEGER	
335.9	O	100	INTEGER	
360.1(1)	I_O	100	INTEGER	
31.3	I	1	ENUMERATION	
31.8	O	1	ENUMERATION	
32.8	I_O	1	ENUMERATION	
44.3	I	10	ENUMERATION	
58.0	O	10	ENUMERATION	
59.0	I_O	10	ENUMERATION	
183.8	I	100	ENUMERATION	
338.6	O	100	ENUMERATION	
334.7(2)	I_O	100	ENUMERATION	
31.4	I	1	ARRAY of INTEGER	1
33.2	O	1	ARRAY of INTEGER	1
33.2	I_O	1	ARRAY of INTEGER	1
31.2	I	1	ARRAY of INTEGER	10
31.1	O	1	ARRAY of INTEGER	10
31.2	I_O	1	ARRAY of INTEGER	10
31.5	I	1	ARRAY of INTEGER	100
31.5	O	1	ARRAY of INTEGER	100
31.5	I_O	1	ARRAY of INTEGER	100
31.0	I	1	ARRAY of INTEGER	10000
31.0	O	1	ARRAY of INTEGER	10000
31.0	I_O	1	ARRAY of INTEGER	10000
31.4	I	1	RECORD of INTEGER	1
33.3	O	1	RECORD of INTEGER	1
33.3	I_O	1	RECORD of INTEGER	1
31.3	I	1	RECORD of INTEGER	100
31.3	O	1	RECORD of INTEGER	100
31.2	I_O	1	RECORD of INTEGER	100

(1) - Results for this test have ranged from 335 to 360 microseconds.
(2) - Results for this test have ranged from 334 to 346 microseconds.

Number of Iterations = 10000 * 10

Dynamic Allocation in a Declarative Region

Time (microsec.)	# Declared	Type Declared	Size o Object
2.3	1	Integer	
7.4	10	Integer	
6.3	100	Integer	
3.9	1	String	1
3.7	1	String	10
4.5	1	String	100
2.2	1	Enumeration	
0.8(1)	10	Enumeration	
18.9(2)	100	Enumeration	
3.6	1	Integer Array	1
5.4	1	Integer Array	10
5.5	1	Integer Array	100
8.2(3)	1	Integer Array	1000
-12.2(4)	1	Integer Array	10000
0.1(5)	1	Integer Array	100000
5.2	1	Integer Array	1000000
30.9	1	1-D Dynamically bounded Array	1
30.7	1	1-D Dynamically bounded Array	10
45.8	1	2-D Dynamically bounded Array	1
45.6	1	2-D Dynamically bounded Array	100
59.2	1	3-D Dynamically bounded Array	1
59.0	1	3-D Dynamically bounded Array	1000
2.1	1	Record of Integer	1
2.7	1	Record of Integer	10
2.6	1	Record of Integer	100

Note: Times reported include any deallocation required upon leaving the
of the declared variables.
(1) - Other runs have indicated that this value is 1.9 microseconds.
(2) - Other runs have indicated that this value is 1.9 microseconds.
(3) - Other runs have indicated that this value is 6.1 microseconds.
(4)(5) - These tests consistently report low values.

Number of Iterations = 10000

Dynamic Allocation with NEW allocator

Time (microsec.)	# Declared	Type Declared	Size of Object
134	1	Integer	1
133	1	Enumeration	1
140	1	Record of Integer	1
211	1	Record of Integer	5
293	1	Record of Integer	10
435	1	Record of Integer	20
1924(1)	1	Record of Integer	50
4847(2)	1	Record of Integer	100
139	1	String	1
157	1	String	10
393	1	String	100
139	1	Integer Array	1
263	1	Integer Array	10
4948(3)	1	Integer Array	100
221	1	1-D Dynamically Bounded Array	1
284	1	1-D Dynamically Bounded Array	10
309	1	2-D Dynamically Bounded Array	1
4782(4)	1	2-D Dynamically Bounded Array	100
326	1	3-D Dynamically Bounded Array	1

Note: Storage is not reclaimed by calling UNCHECKED_DEALLOCATION, this results in excessive disk paging for the noted cases. Runs with fewer iterations result in more accurate results with less precision.

(1) - When run with only 100 iterations this test yields 700 microseconds
(2) - When run with only 100 iterations this test yields 1200 microseconds
(3) - When run with only 100 iterations this test yields 1100 microseconds
(4) - When run with only 100 iterations this test yields 1200 microseconds

```
Number of Iterations = 10000


                    Exception Handler Tests

        Exception raised and handled in a block
           0 uSEC.     User Defined, Not Raised
         345 uSEC.     User Defined
         372 uSEC.     Constraint Error, Implicitly Raised
         356 uSEC.     Constraint Error, Explicitly Raised
          (1)          Numeric Error, Implicitly Raised
         391 uSEC.     Numeric Error, Explicitly Raised
         402 uSEC.     Tasking Error, Explicitly Raised

        Exception raised in a procedure and handled in the calling unit
           0 uSEC.     User Defined, Not Raised
         614 uSEC.     User Defined
         654 uSEC.     Constraint Error, Implicitly Raised
         626 uSEC.     Constraint Error, Explicitly Raised
        2469 uSEC.     Numeric Error, Implicitly Raised
         670 uSEC.     Numeric Error, Explicitly Raised
         671 uSEC.     Tasking Error, Explicitly Raised


(1) - This case is not handled correctly by the compiler.
```

```
Task Elaborate, Activate, and Terminate Time: Declared Object, No Type
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:    19.6 milliseconds.
----------------------------------------------------------------
```

```
Task Elaborate, Activate, and Terminate Time: Declared Object, Task Type
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:    19.6 milliseconds.
----------------------------------------------------------------
```

```
Task Elaborate, Activate, and Terminate Time: NEW Object, Task Type
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:    17.4 milliseconds.
----------------------------------------------------------------
```

```
Rendesvous Time:   No Parameters Passed
Number of Iterations = 100
--------------------------------------------------------------
Task Rendesvous Time:    3.5 milliseconds
--------------------------------------------------------------
```

```
Number of Iterations =  10000



Clock function calling overhead :  568  microseconds
Clock function calling overhead :  572  microseconds
Clock function calling overhead :  569  microseconds
Clock function calling overhead :  569  microseconds
Clock function calling overhead :  575  microseconds
Clock function calling overhead :  578  microseconds
Clock function calling overhead :  585  microseconds
Clock function calling overhead :  577  microseconds
Clock function calling overhead :  563  microseconds
Clock function calling overhead :  575  microseconds
Clock function calling overhead :  563  microseconds
Clock function calling overhead :  569  microseconds
Clock function calling overhead :  567  microseconds
Clock function calling overhead :  575  microseconds
Clock function calling overhead :  568  microseconds
Clock function calling overhead :  573  microseconds
Clock function calling overhead :  571  microseconds
Clock function calling overhead :  562  microseconds
```

```
Number of Iterations = 10000 * 10


             TIME and DURATION math

Microseconds        Operation
    188.5      Time := Var_time + Var_duration
    231.8      Time := Var_time - Var_duration
    226.6      Time := Var_duration + Var_time
    241.9      Time := Var_time - Const_duration
    111.3      Duration := Var_time - Var_time
      7.3      Duration := Var_duration + Var_duration
      7.2      Duration := Var_duration - Var_duration
      7.6      Duration := Var_duration + Const_duration
      7.6      Duration := Var_duration - Const_duration
      7.7      Duration := Const_duration + Var_duration
      7.7      Duration := Const_duration - Var_duration
      1.2      Duration := Const_duration + Const_duration
      1.2      Duration := Const_duration - Const_duration
```

```
Delay Statement Test  -  Minimum Delay Value
Number of Iterations = 10000 * 10
------------------------------------------------------
For case number           1
Desired delay time:     0.000061035156250 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           2
Desired delay time:     0.000122070312500 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           3
Desired delay time:     0.000183105468750 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           4
Desired delay time:     0.000244140625000 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           5
Desired delay time:     0.000305175781250 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           6
Desired delay time:     0.000366210937500 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           7
Desired delay time:     0.000427246093750 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           8
Desired delay time:     0.000488281250000 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           9
Desired delay time:     0.000549316406250 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           10
Desired delay time:     0.000610351562500 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           11
Desired delay time:     0.000671386718750 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           12
Desired delay time:     0.000732421875000 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           13
Desired delay time:     0.000793457031250 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           14
Desired delay time:     0.000854492187500 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           15
Desired delay time:     0.000915527343750 seconds.
Actual delay time:      0.001403808593750 seconds.

For case number           16
Desired delay time:     0.000976562500000 seconds.
Actual delay time:      0.001403808593750 seconds.
```

The following results are consistent with the analysis is given in the report. Requested delays between 1 millisecond and less than 10 milliseconds results in actual delays between 10 milliseconds and 1.01 seconds, that is, the value remaining in the 1 second time slice plus 10 milliseconds. In general, a requested delay of D seconds results in an actual delay between D seconds and D+1 seconds. In all cases the actual delay is always greater than or equal to the requested delay, as required by the LRM.

```
Delay Statement Test
Number of Iterations = 1

--------------------------------------------------
For case number              1
Desired delay time:       0.001037597656250 seconds.
Actual delay time:        1.000000000000000 seconds.

For case number              2
Desired delay time:       0.002075195312500 seconds.
Actual delay time:        0.729980468750000 seconds.

For case number              3
Desired delay time:       0.003112792968750 seconds.
Actual delay time:        0.750000000000000 seconds.

For case number              4
Desired delay time:       0.004150390625000 seconds.
Actual delay time:        0.259948730468750 seconds.

For case number              5
Desired delay time:       0.005187988281250 seconds.
Actual delay time:        0.339965820312500 seconds.

For case number              6
Desired delay time:       0.006225585937500 seconds.
Actual delay time:        0.699951171875000 seconds.

For case number              7
Desired delay time:       0.007263183593750 seconds
Actual delay time:        0.859985351562500 seconds.

For case number              8
Desired delay time:       0.008300781250000 seconds.
Actual delay time:        0.009948730468750 seconds.

For case number              9
Desired delay time:       0.009338378906250 seconds.
Actual delay time:        0.669982910156250 seconds.

For case number             10
Desired delay time:       0.010375976562500 seconds.
Actual delay time:        0.469970703125000 seconds.

For case number             11
Desired delay time:       0.011413574218750 seconds.
Actual delay time:        0.599975585937500 seconds.

For case number             12
Desired delay time:       0.012451171875000 seconds.
Actual delay time:        0.549987792968750 seconds.

For case number             13
Desired delay time:       0.013488769531250 seconds.
Actual delay time:        0.779968261718750 seconds.

For case number             14
Desired delay time:       0.014526367187500 seconds.
Actual delay time:        0.869995117187500 seconds.
```

```
Delay Statement Test
Number of Iterations = 1

-------------------------------------------------------
For case number              1
Desired delay time:         0.097656250000000 seconds.
Actual delay time:          1.000000000000000 seconds.

For case number              2
Desired delay time:         0.195312500000000 seconds.
Actual delay time:          0.719970703125000 seconds.

For case number              3
Desired delay time:         0.292968750000000 seconds.
Actual delay time:          0.750000000000000 seconds.

For case number              4
Desired delay time:         0.390625000000000 seconds.
Actual delay time:          1.259948730468750 seconds.

For case number              5
Desired delay time:         0.488281250000000 seconds.
Actual delay time:          1.339965820312500 seconds.

For case number              6
Desired delay time:         0.585937500000000 seconds.
Actual delay time:          0.709960937500000 seconds.

For case number              7
Desired delay time:         0.683593750000000 seconds.
Actual delay time:          0.859985351562500 seconds.

For case number              8
Desired delay time:         0.781250000000000 seconds.
Actual delay time:          0.949951171875000 seconds.

For case number              9
Desired delay time:         0.878906250000000 seconds.
Actual delay time:          1.729980468750000 seconds.

For case number              10
Desired delay time:         0.976562500000000 seconds.
Actual delay time:          1.469970703125000 seconds.

For case number              11
Desired delay time:         1.074218750000000 seconds.
Actual delay time:          1.599975585937500 seconds.

For case number              12
Desired delay time:         1.171875000000000 seconds.
Actual delay time:          1.549987792968750 seconds.

For case number              13
Desired delay time:         1.269531250000000 seconds.
Actual delay time:          1.779968261718750 seconds.

For case number              14
Desired delay time:         1.367187500000000 seconds.
Actual delay time:          1.869995117187500 seconds.

For case number              15
Desired delay time:         1.464843750000000 seconds.
Actual delay time:          1.589965820312500 seconds.

For case number              16
Desired delay time:         1.562500000000000 seconds.
Actual delay time:          2.269958496093750 seconds.
```

```
Delay Statement Test
Number of Iterations = 1
---------------------------------------------------
For case number              1
Desired delay time:        0.213623046875000 seconds.
Actual delay time:         1.000000000000000 seconds.

For case number              2
Desired delay time:        0.427246093750000 seconds.
Actual delay time:         0.739990234375000 seconds.

For case number              3
Desired delay time:        0.640869140625000 seconds.
Actual delay time:         0.750000000000000 seconds.

For case number              4
Desired delay time:        0.854492187500000 seconds.
Actual delay time:         1.269958496093750 seconds.

For case number              5
Desired delay time:        1.068115234375000 seconds.
Actual delay time:         1.339965820312500 seconds.

For case number              6
Desired delay time:        1.281738281250000 seconds.
Actual delay time:         1.699951171875000 seconds.

For case number              7
Desired delay time:        1.495361328125000 seconds.
Actual delay time:         1.859985351562500 seconds.

For case number              8
Desired delay time:        1.708984375000000 seconds.
Actual delay time:         1.939941406250000 seconds.

For case number              9
Desired delay time:        1.922607421875000 seconds.
Actual delay time:         2.729980468750000 seconds.

For case number              10
Desired delay time:        2.136230468750000 seconds.
Actual delay time:         2.469970703125000 seconds.

For case number              11
Desired delay time:        2.349853515625000 seconds.
Actual delay time:         2.609985351562500 seconds.

For case number              12
Desired delay time:        2.563476562500000 seconds.
Actual delay time:         3.559997558593750 seconds.

For case number              13
Desired delay time:        2.777099609375000 seconds.
Actual delay time:         2.779968261718750 seconds.

For case number              14
Desired delay time:        2.990722656250000 seconds.
Actual delay time:         3.859985351562500 seconds.

For case number              15
Desired delay time:        3.204345703125000 seconds.
Actual delay time:         3.589965820312500 seconds.

For case number              16
Desired delay time:        3.417968750000000 seconds.
Actual delay time:         4.269958496093750 seconds.
```

APPENDIX C

The following pages contain result tables for all of the tests run. These results are for the Verdix Compiler Version 5.1 running with Unix 4.2 bsd on a Vax 11/780. Some values contain explainatory footnotes.

```
Delay Statement Test  -  Minimum Delay Value
Number of Iterations = 100
-------------------------------------------------
For case number            1
Desired delay time:       0.000061035156250 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            2
Desired delay time:       0.000122070312500 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            3
Desired delay time:       0.000183105468750 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            4
Desired delay time:       0.000244140625000 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            5
Desired delay time:       0.000305175781250 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            6
Desired delay time:       0.000366210937500 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            7
Desired delay time:       0.000427246093750 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            8
Desired delay time:       0.000488281250000 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number            9
Desired delay time:       0.000549316406250 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           10
Desired delay time:       0.000610351562500 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           11
Desired delay time:       0.000671386718750 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           12
Desired delay time:       0.000732421875000 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           13
Desired delay time:       0.000793457031250 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           14
Desired delay time:       0.000854492187500 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           15
Desired delay time:       0.000915527343750 seconds.
Actual delay time:        0.019958496093750 seconds.

For case number           16
Desired delay time:       0.000976562500000 seconds.
Actual delay time:        0.019958496093750 seconds.
```

```
Delay Statement Test
Number of Iterations = 100
--------------------------------------------------
For case number              1
Desired delay time:        0.003051757812500  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              2
Desired delay time:        0.003662109375000  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              3
Desired delay time:        0.004272460937500  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              4
Desired delay time:        0.004882812500000  seconds.
Actual delay time:         0.019958496093750  seconds

For case number              5
Desired delay time:        0.005493164062500  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              6
Desired delay time:        0.006103515625000  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              7
Desired delay time.        0.006713867187500  seconds.
Actual delay time.         0.019958496093750  seconds.

For case number              8
Desired delay time:        0.007324218750000  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number              9
Desired delay time:        0.007934570312500  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number             10
Desired delay time:        0.008544921875000  seconds
Actual delay time:         0.019958496093750  seconds

For case number             11
Desired delay time:        0.009155273437500  seconds.
Actual delay time:         0.019958496093750  seconds.

For case number             12
Desired delay time:        0.009765625000000  seconds.
Actual delay time:         0.019958496093750  seconds

For case number             13
Desired delay time:        0.010375976562500  seconds.
Actual delay time:         0.029968261718750  seconds.

For case number             14
Desired delay time:        0.010986328125000  seconds.
Actual delay time:         0.029968261718750  seconds.

For case number             15
Desired delay time:        0.011596679687500  seconds.
Actual delay time:         0.029968261718750  seconds.

For case number             16
Desired delay time:        0.012207031250000  seconds.
Actual delay time.         0.029968261718750  seconds.
```

```
Delay Statement Test
Number of Iterations = 100
----------------------------------------------------------
For case number              1
Desired delay time:        0.030517578125000 seconds.
Actual delay time:         0.049987792968750 seconds.

For case number              2
Desired delay time:        0.036621093750000 seconds.
Actual delay time:         0.049987792968750 seconds.

For case number              3
Desired delay time:        0.042724609375000 seconds.
Actual delay time:         0.059997558593750 seconds.

For case number              4
Desired delay time:        0.048828125000000 seconds.
Actual delay time:         0.059997558593750 seconds.

For case number              5
Desired delay time:        0.054931640625000 seconds.
Actual delay time:         0.069946289062500 seconds.

For case number              6
Desired delay time:        0.061035156250000 seconds.
Actual delay time:         0.079956054687500 seconds.

For case number              7
Desired delay time:        0.067138671875000 seconds.
Actual delay time:         0.079956054687500 seconds.

For case number              8
Desired delay time:        0.073242187500000 seconds.
Actual delay time:         0.089965820312500 seconds.

For case number              9
Desired delay time:        0.079345703125000 seconds.
Actual delay time:         0.089965820312500 seconds.

For case number             10
Desired delay time:        0.085449218750000 seconds.
Actual delay time:         0.099975585937500 seconds.

For case number             11
Desired delay time:        0.091552734375000 seconds.
Actual delay time:         0.109985351562500 seconds.

For case number             12
Desired delay time:        0.097656250000000 seconds.
Actual delay time:         0.109985351562500 seconds.

For case number             13
Desired delay time:        0.103759765625000 seconds.
Actual delay time:         0.119995117187500 seconds.

For case number             14
Desired delay time:        0.109863281250000 seconds.
Actual delay time:         0.119995117187500 seconds.

For case number             15
Desired delay time:        0.115966796875000 seconds.
Actual delay time:         0.129943847656250 seconds.

For case number             16
Desired delay time:        0.122070312500000 seconds.
Actual delay time:         0.139953613281250 seconds.
```

APPENDIX G

The following pages contain result tables for all of the tests run. These results are for the DEC VAX Ada Compiler Version V.1.3 running with VMS 4.4 on a Vax 11/780. Since the results for the delay test are the same for this compiler as they are for V.1.1, please refer to Appendix F for those results.

```
Compiler Time Related Values:
--------------------------------------------

System Tick=          0.009948730468750  Seconds
Duration Small=       0.000061035156250  Seconds
--------------------------------------------
```

Subprogram Overhead (non-generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
-0.8		0		
0.3	I	1	INTEGER	
-0.6	O	1	INTEGER	
-0.8	I_O	1	INTEGER	
2.8	I	10	INTEGER	
12.8	O	10	INTEGER	
17.9	I_O	10	INTEGER	
0.6	I	1	ENUMERATION	
-0.1	O	1	ENUMERATION	
-0.6	I_O	1	ENUMERATION	
2.3	I	10	ENUMERATION	
-0.1	O	10	ENUMERATION	
17.6	I_O	10	ENUMERATION	
-0.8	I	1	ARRAY of INTEGER	1
-0.2	O	1	ARRAY of INTEGER	1
0.1	I_O	1	ARRAY of INTEGER	1
0.4	I	1	ARRAY of INTEGER	10
-0.1	O	1	ARRAY of INTEGER	10
-0.2	I_O	1	ARRAY of INTEGER	10
-0.2	I	1	ARRAY of INTEGER	100
-0.5	O	1	ARRAY of INTEGER	100
0.6	I_O	1	ARRAY of INTEGER	100
-0.1	I	1	ARRAY of INTEGER	10000
0.0	O	1	ARRAY of INTEGER	10000
0.3	I_O	1	ARRAY of INTEGER	10000
0.1	I	1	RECORD of INTEGER	1
-0.1	O	1	RECORD of INTEGER	1
0.2	I_O	1	RECORD of INTEGER	1
-0.2	I	1	RECORD of INTEGER	100
0.4	O	1	RECORD of INTEGER	100
-0.9	I_O	1	RECORD of INTEGER	100
0.2	I	1	UNCONSTRAINED ARRAY	1
0.1	O	1	UNCONSTRAINED ARRAY	1
0.1	I_O	1	UNCONSTRAINED ARRAY	1
-1.8	I	1	UNCONSTRAINED ARRAY	100
0.2	O	1	UNCONSTRAINED ARRAY	100
-1.4	I_O	1	UNCONSTRAINED ARRAY	100
0.5	I	1	UNCONSTRAINED ARRAY	10000
1.1	O	1	UNCONSTRAINED ARRAY	10000
0.3	I_O	1	UNCONSTRAINED ARRAY	10000
0.5	I	1	UNCONSTRAINED RECORD	1
-0.7	O	1	UNCONSTRAINED RECORD	1
-0.4	I_O	1	UNCONSTRAINED RECORD	1
0.6	I	1	UNCONSTRAINED RECORD	100
-0.6	O	1	UNCONSTRAINED RECORD	100
0.0	I_O	1	UNCONSTRAINED RECORD	100

```
                    Subprogram Overhead (inline)
   Number of Iterations = 10000 * 10
-------------------------------------------------------------------------------
    Time     |Direction|# Passed|          Type              | Size of  |
 (microsec.)|  Passed  |in Call |         Passed             |Passed Var|
-------------------------------------------------------------------------------
|    -0.4    |          |       0 |                            |          |
|     1.0    |   I      |       1 |INTEGER                     |          |
|    -0.2    |   O      |       1 |INTEGER                     |          |
|    -0.1    |   I_O    |       1 |INTEGER                     |          |
|     2.4    |   I      |      10 |INTEGER                     |          |
|    10.9    |   O      |      10 |INTEGER                     |          |
|    16.5    |   I_O    |      10 |INTEGER                     |          |
|     0.0    |   I      |       1 |ENUMERATION                 |          |
|    -1.3    |   O      |       1 |ENUMERATION                 |          |
|    -0.9    |   I_O    |       1 |ENUMERATION                 |          |
|     2.1    |   I      |      10 |ENUMERATION                 |          |
|     0.4    |   O      |      10 |ENUMERATION                 |          |
|    18.9    |   I_O    |      10 |ENUMERATION                 |          |
|    -0.5    |   I      |       1 |ARRAY of  INTEGER           |        1 |
|    -0.2    |   O      |       1 |ARRAY of  INTEGER           |        1 |
|     0.2    |   I_O    |       1 |ARRAY of  INTEGER           |        1 |
|     0.0    |   I      |       1 |ARRAY of  INTEGER           |       10 |
|    -0.6    |   O      |       1 |ARRAY of  INTEGER           |       10 |
|     0.0    |   I_O    |       1 |ARRAY of  INTEGER           |       10 |
|    -0.5    |   I      |       1 |ARRAY of  INTEGER           |      100 |
|    -0.4    |   O      |       1 |ARRAY of  INTEGER           |      100 |
|     0.4    |   I_O    |       1 |ARRAY of  INTEGER           |      100 |
|    -0.3    |   I      |       1 |ARRAY of  INTEGER           |    10000 |
|     0.2    |   O      |       1 |ARRAY of  INTEGER           |    10000 |
|     0.5    |   I_O    |       1 |ARRAY of  INTEGER           |    10000 |
|    -0.7    |   I      |       1 |RECORD of  INTEGER          |        1 |
|    -0.1    |   O      |       1 |RECORD of  INTEGER          |        1 |
|     0.5    |   I_O    |       1 |RECORD of  INTEGER          |        1 |
|     0.1    |   I      |       1 |RECORD of  INTEGER          |      100 |
|     0.5    |   O      |       1 |RECORD of  INTEGER          |      100 |
|    -1.3    |   I_O    |       1 |RECORD of  INTEGER          |      100 |
|    -0.2    |   I      |       1 |UNCONSTRAINED ARRAY         |        1 |
|    -1.8    |   O      |       1 |UNCONSTRAINED ARRAY         |        1 |
|     0.5    |   I_O    |       1 |UNCONSTRAINED ARRAY         |        1 |
|    -0.9    |   I      |       1 |UNCONSTRAINED ARRAY         |      100 |
|    -0.4    |   O      |       1 |UNCONSTRAINED ARRAY         |      100 |
|    -1.0    |   I_O    |       1 |UNCONSTRAINED ARRAY         |      100 |
|    -1.1    |   I      |       1 |UNCONSTRAINED ARRAY         |    10000 |
|     0.3    |   O      |       1 |UNCONSTRAINED ARRAY         |    10000 |
|     0.7    |   I_O    |       1 |UNCONSTRAINED ARRAY         |    10000 |
|     1.5    |   I      |       1 |UNCONSTRAINED RECORD        |        1 |
|     0.5    |   O      |       1 |UNCONSTRAINED RECORD        |        1 |
|    -0.6    |   I_O    |       1 |UNCONSTRAINED RECORD        |        1 |
|     1.7    |   I      |       1 |UNCONSTRAINED RECORD        |      100 |
|    -0.1    |   O      |       1 |UNCONSTRAINED RECORD        |      100 |
|    -1.6    |   I_O    |       1 |UNCONSTRAINED RECORD        |      100 |
```

Subprogram Overhead (generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
0.9		0		
0.2	I	1	INTEGER	
1.1	O	1	INTEGER	
2.8	I_O	1	INTEGER	
7.3	I	10	INTEGER	
8.5	O	10	INTEGER	
15.7	I_O	10	INTEGER	
-0.1	I	1	ENUMERATION	
0.1	O	1	ENUMERATION	
0.2	I_O	1	ENUMERATION	
7.4	I	10	ENUMERATION	
8.9	O	10	ENUMERATION	
15.9	I_O	10	ENUMERATION	
-0.5	I	1	ARRAY of INTEGER	1
0.0	O	1	ARRAY of INTEGER	1
0.4	I_O	1	ARRAY of INTEGER	1
-1.2	I	1	ARRAY of INTEGER	10
0.1	O	1	ARRAY of INTEGER	10
0.2	I_O	1	ARRAY of INTEGER	10
-0.1	I	1	ARRAY of INTEGER	100
-0.1	O	1	ARRAY of INTEGER	100
-0.1	I_O	1	ARRAY of INTEGER	100
-0.3	I	1	RECORD of INTEGER	1
-0.5	O	1	RECORD of INTEGER	1
-0.2	I_O	1	RECORD of INTEGER	1
-0.1	I	1	RECORD of INTEGER	100
0.0	O	1	RECORD of INTEGER	100
0.9	I_O	1	RECORD of INTEGER	100

Subproghram Overhead (cross package, non-generic)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
27.0		0		
30.3	I	1	INTEGER	
33.3	O	1	INTEGER	
30.8	I_O	1	INTEGER	
41.4	I	10	INTEGER	
72.0	O	10	INTEGER	
105.9	I_O	10	INTEGER	
30.3	I	1	ENUMERATION	
33.4	O	1	ENUMERATION	
28.6	I_O	1	ENUMERATION	
41.5	I	10	ENUMERATION	
72.1	O	10	ENUMERATION	
108.1	I_O	10	ENUMERATION	
30.7	I	1	ARRAY of INTEGER	1
30.9	O	1	ARRAY of INTEGER	1
27.3	I_O	1	ARRAY of INTEGER	1
30.3	I	1	ARRAY of INTEGER	10
30.7	O	1	ARRAY of INTEGER	10
27.6	I_O	1	ARRAY of INTEGER	10
30.9	I	1	ARRAY of INTEGER	100
31.2	O	1	ARRAY of INTEGER	100
28.0	I_O	1	ARRAY of INTEGER	100
31.3	I	1	ARRAY of INTEGER	10000
31.6	O	1	ARRAY of INTEGER	10000
27.6	I_O	1	ARRAY of INTEGER	10000
31.8	I	1	RECORD of INTEGER	1
32.1	O	1	RECORD of INTEGER	1
29.2	I_O	1	RECORD of INTEGER	1
32.1	I	1	RECORD of INTEGER	100
31.8	O	1	RECORD of INTEGER	100
28.9	I_O	1	RECORD of INTEGER	100
42.9	I	1	UNCONSTRAINED ARRAY	1
42.6	O	1	UNCONSTRAINED ARRAY	1
39.4	I_O	1	UNCONSTRAINED ARRAY	1
42.2	I	1	UNCONSTRAINED ARRAY	100
42.2	O	1	UNCONSTRAINED ARRAY	100
39.5	I_O	1	UNCONSTRAINED ARRAY	100
42.3	I	1	UNCONSTRAINED ARRAY	10000
42.3	O	1	UNCONSTRAINED ARRAY	10000
39.5	I_O	1	UNCONSTRAINED ARRAY	10000
34.0	I	1	UNCONSTRAINED RECORD	1
31.7	O	1	UNCONSTRAINED RECORD	1
28.4	I_O	1	UNCONSTRAINED RECORD	1
30.7	I	1	UNCONSTRAINED RECORD	100
31.2	O	1	UNCONSTRAINED RECORD	100
28.4	I_O	1	UNCONSTRAINED RECORD	100

Subprogram Overhead (generic, cross package)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
15.0		0		
20.3	I	1	INTEGER	
22.7	O	1	INTEGER	
24.4	I_O	1	INTEGER	
64.4	I	10	INTEGER	
79.4	O	10	INTEGER	
104.4	I_O	10	INTEGER	
20.3	I	1	ENUMERATION	
22.8	O	1	ENUMERATION	
24.9	I_O	1	ENUMERATION	
64.6	I	10	ENUMERATION	
79.8	O	10	ENUMERATION	
103.6	I_O	10	ENUMERATION	
20.9	I	1	ARRAY of INTEGER	1
21.1	O	1	ARRAY of INTEGER	1
24.0	I_O	1	ARRAY of INTEGER	1
45.2	I	1	ARRAY of INTEGER	10
43.3	O	1	ARRAY of INTEGER	10
61.0	I_O	1	ARRAY of INTEGER	10
157.6	I	1	ARRAY of INTEGER	100
159.7	O	1	ARRAY of INTEGER	100
288.2	I_O	1	ARRAY of INTEGER	100
21585.6	I	1	ARRAY of INTEGER	10000
21552.2	O	1	ARRAY of INTEGER	10000
42904.3	I_O	1	ARRAY of INTEGER	10000
20.8	I	1	RECORD of INTEGER	1
22.0	O	1	RECORD of INTEGER	1
25.2	I_O	1	RECORD of INTEGER	1
153.8	I	1	RECORD of INTEGER	100
153.3	O	1	RECORD of INTEGER	100
282.5	I_O	1	RECORD of INTEGER	100

Number of Iterations = 10000 * 10

Dynamic Allocation in a Declarative Region

Time (microsec.)	# Declared	Type Declared	Size of Object
0.5	1	Integer	
1.5	10	Integer	
1.0	1	String	1
0.4	1	String	10
1.6	1	String	100
4.0	1	Enumeration	
1.0	10	Enumeration	
1.1	1	Integer Array	1
1.1	1	Integer Array	10
0.5	1	Integer Array	100
1.7	1	Integer Array	1000
1.5	1	Integer Array	10000
8.3	1	Integer Array	100000
17.8	1	1-D Dynamically bounded Array	1
15.2	1	1-D Dynamically bounded Array	10
30.9	1	2-D Dynamically bounded Array	1
21.1	1	2-D Dynamically bounded Array	100
47.7	1	3-D Dynamically bounded Array	1
45.8	1	3-D Dynamically bounded Array	1000
0.9	1	Record of Integer	1
0.7	1	Record of Integer	10
1.8	1	Record of Integer	100

Note: Times reported include any deallocation required upon leaving the scope of the declared variables.

Number of Iterations = 1000

Dynamic Allocation with NEW allocator

Time (microsec.)	# Declared	Type Declared	Size of Object
250	1	Integer	1
240	1	Enumeration	1
250	1	Record of Integer	1
240	1	Record of Integer	5
250	1	Record of Integer	10
250	1	Record of Integer	20
250	1	Record of Integer	50
250	1	Record of Integer	100
260	1	String	1
260	1	String	10
260	1	String	100
250	1	Integer Array	1
250	1	Integer Array	10
240	1	Integer Array	100
230	1	Integer Array	1000
290	1	1-D Dynamically Bounded Array	1
300	1	1-D Dynamically Bounded Array	10
300	1	2-D Dynamically Bounded Array	1
280	1	2-D Dynamically Bounded Array	100
370	1	3-D Dynamically Bounded Array	1
370	1	3-D Dynamically Bounded Array	1000

```
   Number of Iterations = 10000


                    Exception Handler Tests
                    =======================

          Exception raised and handled in a block
                3 uSEC.     User Defined, Not Raised
              414 uSEC.     User Defined
              532 uSEC.     Constraint Error, Implicitly Raised
              541 uSEC.     Constraint Error, Explicitly Raised
              599 uSEC.     Numeric Error, Implicitly Raised
              541 uSEC.     Numeric Error, Explicitly Raised
              524 uSEC.     Tasking Error, Explicitly Raised

          Exception raised in a procedure and handled in the calling unit
               12 uSEC.     User Defined, Not Raised
              482 uSEC.     User Defined
              619 uSEC.     Constraint Error, Implicitly Raised
              598 uSEC.     Constraint Error, Explicitly Raised
              597 uSEC.     Numeric Error, Implicitly Raised
              605 uSEC.     Numeric Error, Explicitly Raised
              593 uSEC.     Tasking Error, Explicitly Raised
```

```
Task Elaborate, Activate, and Terminate Time: Declared Object, No Ty
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:     6.4 milliseconds.
------------------------------------------------------------
```

```
Task Elaborate, Activate, and Terminate Time: Declared Object, Task
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:     6.6 milliseconds.
------------------------------------------------------------
```

```
Task Elaborate, Activate, and Terminate Time: NEW Object, Task Type
Number of Iterations = 100

For test number          1
Task elaborate, activate, terminate time:     7.8 milliseconds.
------------------------------------------------------------
```

```
Rendezvous Time:   No Parameters Passed
Number of Iterations = 100
-----------------------------------------------------------------
Task Rendezvous Time:    1.1 milliseconds
-----------------------------------------------------------------
```

```
Number of Iterations =  10000

Clock function calling overhead :  89 microseconds.
```

```
Number of Iterations = 10000 * 10


          TIME and DURATION math
          ======================
Microseconds        Operation
    91.2       Time := Var_time + Var_duration
    91.6       Time := Var_time - Var_duration
    90.8       Time := Var_duration + Var_time
    92.0       Time := Var_time - Const_duration
    94.3       Duration := Var_time - Var_time
     0.4       Duration := Var_duration + Var_duration
     0.8       Duration := Var_duration - Var_duration
     0.9       Duration := Var_duration + Const_duration
     0.8       Duration := Var_duration - Const_duration
     0.9       Duration := Const_duration + Var_duration
     1.1       Duration := Const_duration - Var_duration
     0.9       Duration := Const_duration + Const_duration
     0.8       Duration := Const_duration - Const_duration
```