RSD-TR-6-86

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

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

January 1986

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robqt Systems Division

COLLEGE OF ENGINEERING
THE UNIVERSITY OF MICHIGAN
ANN ARBOR, MICHIGAN 48109-1109

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

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

ensn

UMR 1198

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, will be 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 benchmarks tests for the Verdix version 4.06 and version 5.1 compilers for a VAX UNIX system. It should be noted that this version of the compiler is 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, it was the principal Ada compiler 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, if implemented, 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

```
$T_1$ := CLOCK;
while I < N loop
        control functions;
        DO_SEPARATE_PROC_F ; -- the function whose time is being measured
        INCREMENT(I);                                              (L1)
end loop;
$T_2$ := CLOCK;
$T_m$ := $T_2 - T_1$;
```

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 τ 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

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

where $\lfloor x \rfloor$ is the "floor" function (the largest integer less than or equal to x), τ_c is the nominal time required to perform the CLOCK function, and τ_v is a variable indicating a (hopefully) small random variation in the time required to perform the CLOCK function. Since a difference of CLOCK measurements will be used, τ_c will subtract out of the equations to be developed and can be ignored. It is assumed in all of the equations that follow that τ_v is small in comparison to τ and can also be ignored. In any application, however, this assumption must be verified. One of the tests described 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

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

where

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

Then T_F is given by

$$T_F = \frac{T_m}{N} - T_0 \pm \frac{\delta \cdot \tau}{N} \ . \tag{3}$$

Thus, the accuracy of the measurement is determined by

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

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

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

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

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

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

$$\Delta T_m(k) = T_m(k+1) - T_m(k) = n \cdot \tau + \{\lfloor (k+1) \cdot \delta \rfloor - \lfloor k \cdot \delta \rfloor\}\tau \tag{8}$$

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

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

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

$$\Delta\ T_m(k) - \Delta T_m(k-1) = 0,\ \tau \text{ or } -\tau \tag{10}$$

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

$$\tau, -\tau, 0, ..., 0, \tau, -\tau, ..., 0, \tau, -\tau, 0, \cdots \tag{11a}$$

or

$$-r,\ r,\ 0,\ \ldots,0,\ -r,\ r,\ 0,\ldots,\ 0 \cdots \quad . \tag{11b}$$

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

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

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

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 processors 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 require a constant T_c 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_{os}, 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

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

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

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

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

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 $-1 < \epsilon_c, \epsilon_{cf} < 1$. Then, it can be found that

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

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

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

The two right hand terms in Eq. (20) can be made arbitrary small by choosing N sufficiently large. The effect of β shows that the results previously obtained in Eq. (3) are pessimistic and that a correction can be applied if T_p and T_e can be determined.

Estimates of T_p and T_e 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_e satisfies the relation $T_e >> r$, and that $T_p = m \cdot r$ for $m >> 1$, and that δ in equation (5) is zero. The latter assumption means that the contribution to the second difference from the resolution time, r, itself is also zero, and the following analysis will reflect only the effects of T_e. From a filtering point of view, the time measurements are simply a staircase input to simple second difference filter. The output string, then, is just

$$0, 0..0, T_e, -T_e, 0, ..., 0, T_e, -T_e, 0 \cdots . \qquad (21)$$

This directly yields T_e, 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_e by $\pm r$. Further, if T_e is not an integral multiple of r, the values in the sequence will only be within r of T_e. If $T_e >> r$, reasonable estimates of the parameters should still be obtainable. Theoretically, it is possible to derive the precise value of T_e based upon the number periods in (21) between fluctuations of size r in the values. In practice this will be difficult to detect because of the length of sequence required and the distortion from the $(r,-r)$ occurrence as in (11).

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

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

Actual tests with this approach revealed another difficulty. Some implementations of the CLOCK function involve the dynamic allocation of records, which in turn may involve the invocation of a run-time system function. As will be discussed in section 5.3, the time required to perform this operation can vary quite widely. This variation in storage allocation time will give the appearance of operating system overhead. To avoid these problems, the Ada CLOCK function should not be used for tests to determine the operating system overhead. Instead, an implementation dependent subprogram is required which can read the system timer without invoking any variable time system functions such as storage allocation. Such a system dependent subprogram was written and used in our tests. It should be noted, however, that for all of the other tests to be described, the CLOCK function is evaluated only at the beginning and the end of a loop iterated a large number of times, and the effect of the dynamic storage allocation is effectively eliminated, as shown in eq. (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 Eq. (20) was based upon a periodically occurring function which always 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 the assumptions and improve the accuracy of the results obtained.

The distribution of the estimates can be observed by running a repeated set of trials. 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 Section 3, specific details are given. Listings of all the benchmark tests are given in the Appendix.

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 percent of a typical Ada program as implemented in the iMAX 432 system was subprogram calls [7]. Shimasaki, et al., found the figure to be 40.3 percent 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 instantations 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 instantations 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 time are determined for 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. The rendezvous involves at least two context switches to complete: 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 Figure 1. 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 loop 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 Figure 2. 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 DELAY 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. 3. 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. Some cleverness is required in selecting the values of DEL to use, e.g., a coarse to fine research 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 instrinsically 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 responsbility. 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 virutal 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 section 3.1 can be used to measure the relevant times which occur.

For one test, a large array of pointers to a sizeable 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 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 occurence).

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 equation (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 two Verdix VAX compilers, versions 4.06 and 5.1, running with UNIX 4.2 BSD. All tests were run on a VAX 11/780 with all other user and daemon processes disabled (except for the swapper and page daemon, which can never be disabled). We ran the tests described in Sec. 3 to determine the operating system overhead injected into measurements. The components to overhead individually required significantly less than the resolution of the time measurement, r. Thus, as indicated in Sec. 3.3, it is 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 estimate 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. In other words, the results presented are pessimistic and could be modified via eq. (20), where we have estimated the β parameter to be 0.05.

The number of iterations used in the test and control loops was either 100,000 or 100 in order to produce results theoretically accurate to the nearest tenth of a microsecond or tenth of a millisecond (except where noted). The tests provided values that were actually repeatable to within 5 percent (except where noted).

Highlights of the test results obtained are given here. Full tabular listings appear in the Appendix A for version 4.06 and Appendix B for version 5.1. In the discussion that follows, numbers in parenthesis refer to version 5.1 results. The other numbers refer to version 4.06 results, except where noted.

5.1. Subprogram Overhead

The results for these tests show that the value for subprogram overhead is 28(17) microseconds when no parameters are passed. For version 4.06, the same value was obtained in the cases where the INLINE **pragma** was used, i.e., INLINE is not implemented. For version 5.1 however, the INLINE pragma is implemented and the overhead is reduced to the time needed to copy parameters and check constraints where applicable. Surprisingly, cross-package calls were faster than the intra-package calls by 10 microseconds for version 4.06 e.g., 18 microseconds instead of 28 for the basic parameterless call. When the subprograms called were instantiations of generics, the subprogram calling time was greater by 2.5(1.5) microseconds, i.e. 30.5(18.5) microseconds when the call was in same package. In the case where a subprogram in a generic package calls another subprogram in a separate generic package, the calling overhead was 26(22) microseconds. It appears that the initial version of the compiler handled intra-package calls in an inefficient manner, making them slower than cross-package calls. This situation has been corrected in the newer version of the compiler.

In general, for both compiler versions, the values in each of the tests were greater by 1.5 to 2 microseconds for each variable integer or scalar passed **in** and 3 to 4 microseconds for parameters passed in modes **out** and **in out**. For version 4.06, additional times to pass arrays and records of one integer are 5 and 4 microseconds respectively for modes **out** and **in out**. In version 5.1, these times are the same as the times to pass one INTEGER. This is due to the fact that these parameters are passed by copy. Arrays and Records of 10 or more integers are passed by reference, with calling overhead constant for all three modes. In the case of unconstrained arrays, calling overhead is increased by approximately 1.5(2) microseconds over the time to pass arrays by reference. This is because constraint information must also be passed at run time.

5.2. Dynamic Allocation of Objects

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

The declarative region tests reveal that the allocation times are independent of the size of the object declared for small objects. INTEGERS required 2(0) microseconds regardless of the number of INTEGERS declared. Similarly STRINGs require 3(0) microseconds and statically bounded arrays require 4(1) microseconds. Dynamic arrays require 30 microseconds for the first dimension plus 15 microseconds for each additional dimension.

The **new** allocator tests suffer from accuracy problems due to the underlying memory management mechanism. For the version 4.06 compiler, the storage allocated is never deallocated. Thus, as the amount of storage allocated increases toward the real storage size, the operating system begins to swap memory pages onto disk. The paging time is sufficient to distort the test results. To eliminate this difficulty, a sequence of pretests were run to determine, for our system, 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 and accurate results were still obtained.

For the version 4.06 compiler, the allocation of a single INTEGER required 133 ± 10 microseconds. The times reported for the allocation of records of integers start at 140 microseconds for a record of one integer and grows linearly to a value of 1200 microseconds for a record of 100 integers.

The allocation times reported for the version 5.1 compiler are higher than those reported for the version 4.06 compiler since memory is actually reclaimed by UNCHECKED_DEALLOCATION. The version 5.1 compiler yielded 227 microseconds for the allocation of a single INTEGER, and 230 microseconds for the allocation of a single enumeration type. Allocation times for records increased slightly as the size of the record was increased.

An interesting insight to the storage management process was obtained from our original attempts to estimate the operating system overhead using the Ada CLOCK function. As noted earlier we found that this function involved the dynamic allocation of TIME records. Moreover, the time required for the allocation was not constant. The storage allocator initially obtains a chunk of memory from the heap which, for small objects, might be considerably larger than the size of the object being allocated. Subsequent allocations are made from the leftover memory in this chunk until it is used up, at which time the allocator again gets a chunk of memory from the heap. The time to allocate from the chunk is only a few 10's of microseconds, while the time to obtain the chunk from the heap is on the order of 3 milliseconds. Thus, the time to allocate any one object can be quite variable, and the numbers sited in the previous paragraph are only average figures. This effect will principally occur with data objects which are small compared to the size of the chunk of storage obtained from the heap.

The possibility of storage allocation occasionally taking a long time due to the need to access the heap can have an innocuously devastating effect on real-time programs which is difficult to discern from benchmark tests. The difficulty arises from implementation supplied procedures or functions which internally allocate records. CLOCK is the prime example. It will be used in many, if not most, real-time scheduling loops. If it allocates a record, as has been the case on every compiler we have tested to date, it will sometimes result in a new allocation from the heap which will take a much longer time (possibly two orders of magnitude) and very possibly cause a major perturbation in the timing. Moreover, since it won'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 in a way which might cause the allocation to come from the heap.

5.3. Exceptions

The exception handling tests are divided into two sets. The first set of tests raise an exception within a declare block; the exception is then handled by a null handler at the end of the block. The second set of tests raise an exception 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.

Exceptions raised and handled within a block, result in times of 345(315) to 402(372) microseconds, depending on the exception and how it was raised. For example CONSTRAINT_ERROR requires 356(325) microseconds when raised explicitly and 372(343) microseconds when raised by violating a subtype range constraint. A block with an exception handler and no exception raised results in no additional overhead.

For exceptions raised within a called procedure and handled within the calling block we found times of 614(544) to 671(613) microseconds. The difference between these times and the times for exceptions fielded within a block reflect the time required to propagate an exception from a procedure to the calling block.

The propagation time for NUMERIC_ERROR is 2469(2551) microseconds when raised by the overflow of an INTEGER. This unusually large time is probably due to the UNIX operating system regaining control to handle the hardware trap.

5.4. Task Elaboration, Activation, and Termination

This test was run for the three different types of task activation, explained in section 4.5. As with most of the benchmarks described in this paper, the test was embedded in a loop executed a large number of times to improve the accuracy of the measurement. Because of the storage allocation involved in elaborating and activating a task, numerous activations cause the program size to grow, and paging activity may again distort the results obtained. In order to avoid this problem, the test was run with several different numbers of iterations to determine the order of magnitude of the metric. Then, the number of iterations was reduced to a level where further reduction had no effect on the measured time. At this point, thrashing is avoided. For this test, 100 iterations did not introduce thrashing and provided accuracy to the nearest tenth of a millisecond.

Another possible pitfall in making this measurement is the inclusion of excessive operating system time in allocating storage. As mentioned in section 5.2, storage allocation times are variable depending on the type of allocation that has to be done. Since many tasks are created in this benchmark, the probability that a large chunk of storage needs to be allocated is increased. In UNIX, calls are made to the system routines MALLOC and SBRK. MALLOC acquires storage for the program without much overhead if it is available. If it is not, MALLOC calls SBRK to allocate a large chunk of storage. In order to avoid including time to call SBRK in the benchmark, an initial call is made to MALLOC requesting a very large chunk of storage, implicitly forcing a call to SBRK. This is done outside of the timed loop. This storage is then deallocated by calling the UNIX routine FREE. At this point, the storage allocated by MALLOC and then released by FREE is available to be acquired in future calls to MALLOC, and SBRK will not need to be called. The calls to UNIX storage allocation routines were made possible by the INTERFACE pragma, which is supported by both compiler versions.

The time required to elaborate, activate and terminate a task declared, either directly or via a task type, in a block is approximately 19.6 milliseconds for version 4.06, and approximately 20.4 milliseconds for version 5.1. Activation via the **new** allocator reduced the composite time to approximately 17.4 milliseconds for both compiler versions.

5.5. Task Synchronization

The test here was rather straightforward. The test involved entering a block where a task is activated and a subprogram is called that executes a rendezvous with that task repeatedly in a loop. The control for this test is of the same structure, except that the loop is iterated with no rendezvous. The time for a rendezvous was determined to be 3.5 milliseconds for both compiler versions. Entry calls with parameters showed that the additional time to pass parameters was negligible.

5.6. Clock Function Overhead

For version 4.06, the average time required to call the CLOCK function is 571 microseconds. The actual values varied between 562 microseconds and 585 microseconds. This variation is due to the fact that the CLOCK function must allocate a record to hold the results returned and this allocation time can vary somewhat. In the case of version 5.1, though, the average CLOCK function calling overhead is 3.5 milliseconds, approximately 6 times greater than the value for version 4.06. This increase in overhead is due to a change in the data structure for objects of type TIME and an increase in 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 accounted for and the time is normalized with respect to Grenwich Meridian Time by function calls to Ada subprograms in the CALENDAR package. 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. Besides the increase in overhead due to procedure and function calls, some of the called subprograms perform memory allocation, increasing the total overhead even more. We were able to determine this information about the CLOCK function by examining the source code of the body of the CALENDAR package.

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. A few observations follow:

(1) Constant expressions appear to be evaluated at compile time. The two cases where DURATION constants are added or subtracted required 1.2(1.1,1.6) microseconds, which is significantly less time than the same operations applied to variables.

(2) Variable addition and subtraction involving only the type DURATION as operands and result type take 7.5(6.3 to 9.0) microseconds.

(3) The case of subtracting two TIMEs to return a DURATION requires 111(49) microseconds.

(4) Addition or subtraction of a DURATION to/from a TIME, returning type TIME, requires approximately 200(750) microseconds.

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 two versions of the compiler for expressions involving type TIME is due to a change in the record used to represent TIME. The extra time required to return an object of type TIME is probably associated with the dynamic allocation of a record to hold the result, which takes on the order of 100 microseconds.

5.8. Delay and Scheduling Measurements

This test involved the measurement of time elapsed during the execution of a **delay** statement. The results discussed here apply to both compiler versions.

Using the usual iteration method, a minimum delay value of 1.4 milliseconds was detected. This delay resulted for requested delays from zero to 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. 3.

The actual delay values in other cases were more difficult to isolate, due to the nature of the scheduling system. Verdix uses fixed interval delay scheduling with a delay value of 1 second. What this did to the test was, for a requested delay of 1 millisecond, delay for the remainder of the one second time slice on the first iteration, and delay for a full second on each subsequent iteration. Thus, over a large number of iterations, the average time tended toward 1 second. As the requested delay was increased, the staircase function of Fig. 3 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.

To further test this hypothesis on scheduler behavior, 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 timed loop 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. In general, a requested delay of d seconds results in an actual delay between d seconds and slightly more than 1+d seconds. In all cases the actual delay is always greater than or equal to the requested delay, as required by the LRM.

5.9. Storage Deallocation and Garbage Collection

The storage deallocation tests provide an insight to what type of deallocation facilities are provided for objects declared dynamically with the **new** allocator. The object used for allocation throughout this test is 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 was the size at which

STORAGE_ERROR was raised by loop 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.

For the version 5.1 compiler we found that the run-time system does not perform garbage collection, and calls to UNCHECKED_DEALLOCATION does reclaim storage for scalar types, records, strings and statically bounded array types. Storage is not reclaimed for unconstrained array types.

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 incoroporation of pragma INLINE, improved storage managment facilities, higher speed operations with respect to TIME, and a reduction in tasking 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.

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 Thierault for numerous late nights in helping run the benchmark tests.

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, 1976.

[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. Jalice, "Comparative performance of cobol vs pl/1 programs," *Computer Performance Evaluation Users Group 16th Meeting*, Oct. 1980.

[5] J. Squire, "Performance issues workshop," *ACM SIGADA Users Committee Performance Issues Working Group*, July 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.*, vol. V, no. 2, Sept./Oct. 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.

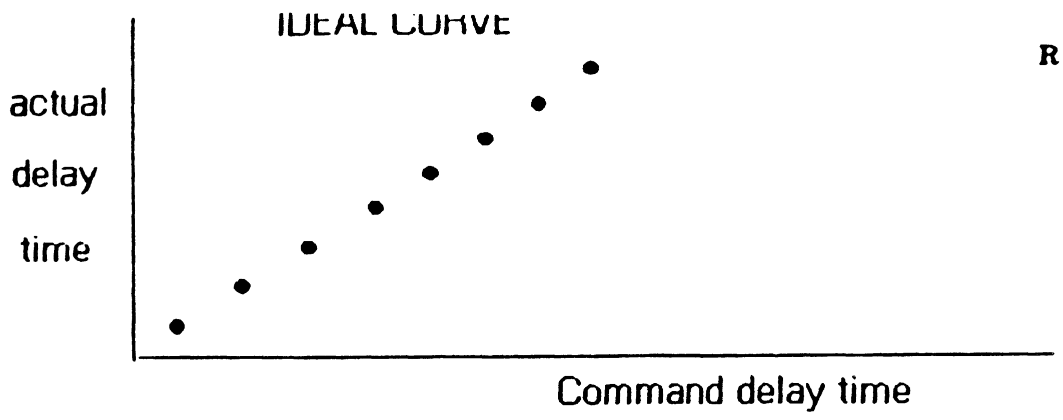

Figure 1

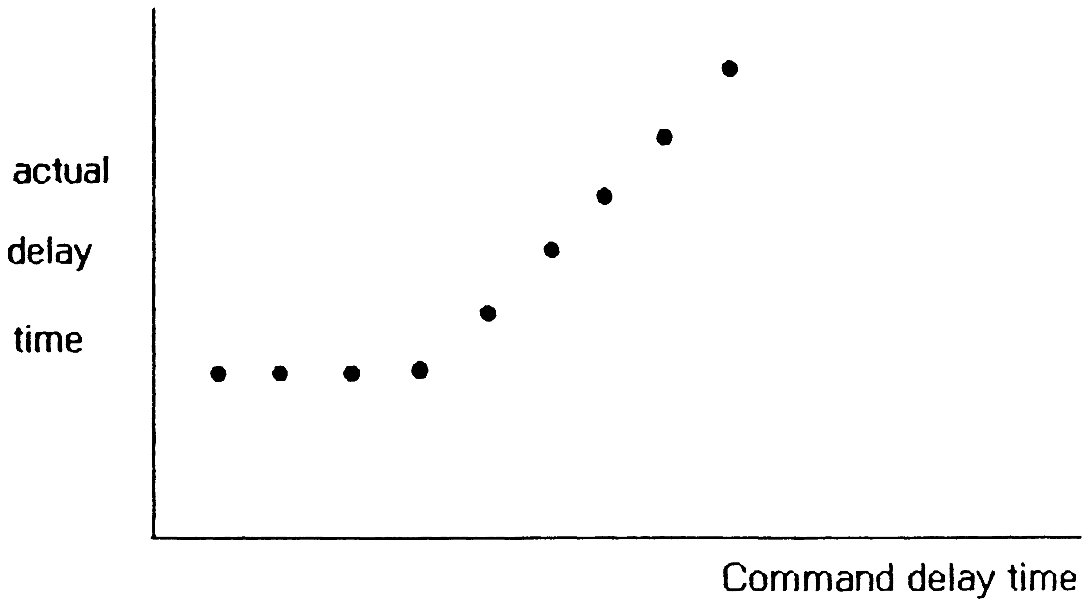

Figure 2

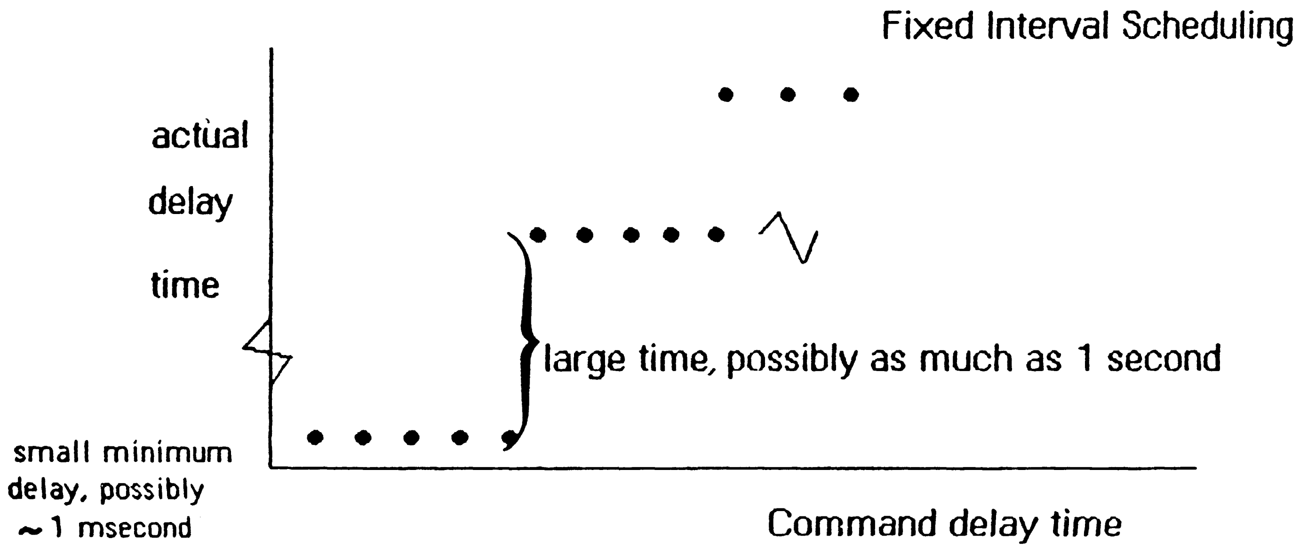

Figure 3

APPENDIX A

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 of 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 sc
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 = 1000

Dynamic Allocation with NEW allocator

Time (microsec.)	# Declared	Type Declared	Size of Object
134.0	1	Integer	1
133.0	1	Enumeration	1
140.0	1	Record of Integer	1
211.0	1	Record of Integer	5
293.0	1	Record of Integer	10
435.0	1	Record of Integer	20
1924.0(1)	1	Record of Integer	50
4847.0(2)	1	Record of Integer	100
139.0	1	String	1
157.0	1	String	10
393.0	1	String	100
139.0	1	Integer Array	1
263.0	1	Integer Array	10
4948.0(3)	1	Integer Array	100
221.0	1	1-D Dynamically Bounded Array	1
284.0	1	1-D Dynamically Bounded Array	10
309.0	1	2-D Dynamically Bounded Array	1
4782.0(4)	1	2-D Dynamically Bounded Array	100
326.0	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
         0.0 uSEC.     User Defined, Not Raised
       345.0 uSEC.     User Defined
       372.0 uSEC.     Constraint Error, Implicitly Raised
       356.0 uSEC.     Constraint Error, Explicitly Raised
          (1)          Numeric Error, Implicitly Raised
       391.0 uSEC.     Numeric Error, Explicitly Raised
       402.0 uSEC.     Tasking Error, Explicitly Raised

      Exception raised in a procedure and handled in the calling unit
         0.0 uSEC.     User Defined, Not Raised
       614.0 uSEC.     User Defined
       654.0 uSEC.     Constraint Error, Implicitly Raised
       626.0 uSEC.     Constraint Error, Explicitly Raised
      2469.0 uSEC.     Numeric Error, Implicitly Raised
       670.0 uSEC.     Numeric Error, Explicitly Raised
       671.0 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.
----------------------------------------------------------------
```

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

```
Number of Iterations =  10000


Clock function calling overhead :  568.00  microseconds
Clock function calling overhead :  572.00  microseconds
Clock function calling overhead :  569.00  microseconds
Clock function calling overhead :  569.00  microseconds
Clock function calling overhead :  575.00  microseconds
Clock function calling overhead :  578.00  microseconds
Clock function calling overhead :  585.00  microseconds
Clock function calling overhead :  577.00  microseconds
Clock function calling overhead :  563.00  microseconds
Clock function calling overhead :  575.00  microseconds
Clock function calling overhead :  563.00  microseconds
Clock function calling overhead :  569.00  microseconds
Clock function calling overhead :  567.00  microseconds
Clock function calling overhead :  575.00  microseconds
Clock function calling overhead :  568.00  microseconds
Clock function calling overhead :  573.00  microseconds
Clock function calling overhead :  571.00  microseconds
Clock function calling overhead :  562.00  microseconds
```

```
Number of Iterations = 10000 * 10



              TIME and DURATION math
              ======================
Microseconds  Operation
   188.50     Time := var_time + var_duration
   231.80     Time := Var_time - Var_duration
   226.60     Time := Var_duration + Var_time
   241.90     Time := Var_time - Const_duration
   111.30     Duration := Var_time - Var_time
     7.30     Duration := var_duration + var_duration
     7.20     Duration := Var_duration - Var_duration
     7.60     Duration := Var_duration + Const_duration
     7.60     Duration := Var_duration - Const_duration
     7.70     Duration := Const_duration + Var_duration
     7.70     Duration := Const_duration - Var_duration
     1.20     Duration := Const_duration + Const_duration
     1.20     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 given in the report. 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. 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 B

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.

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

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

-------------------------------------------------------------------------------
   Time    |Direction|# Passed|          Type           |  Size of |
(microsec.)| Passed  |in Call |         Passed          |Passed Var|
-------------------------------------------------------------------------------
|   16.8   |         |      0 |                         |          |
|   16.8   | I       |      1 | INTEGER                 |          |
|   20.1   | O       |      1 | INTEGER                 |          |
|   19.2   | I_O     |      1 | INTEGER                 |          |
|   29.3   | I       |     10 | INTEGER                 |          |
|   42.2   | O       |     10 | INTEGER                 |          |
|   45.3   | I_O     |     10 | INTEGER                 |          |
|  168.4   | I       |    100 | INTEGER                 |          |
|  325.1   | O       |    100 | INTEGER                 |          |
|  329.3   | I_O     |    100 | INTEGER                 |          |
|   16.5   | I       |      1 |ENUMERATION              |          |
|   17.6   | O       |      1 |ENUMERATION              |          |
|   18.5   | I_O     |      1 |ENUMERATION              |          |
|   29.7   | I       |     10 |ENUMERATION              |          |
|   56.9   | O       |     10 |ENUMERATION              |          |
|   59.5   | I_O     |     10 |ENUMERATION              |          |
|  168.3   | I       |    100 |ENUMERATION              |          |
|  320.0   | O       |    100 |ENUMERATION              |          |
|  326.6   | I_O     |    100 |ENUMERATION              |          |
|   16.9   | I       |      1 |ARRAY of INTEGER         |       1  |
|   20.2   | O       |      1 |ARRAY of INTEGER         |       1  |
|   19.2   | I_O     |      1 |ARRAY of INTEGER         |       1  |
|   19.2   | I       |      1 |ARRAY of INTEGER         |      10  |
|   17.7   | O       |      1 |ARRAY of INTEGER         |      10  |
|   18.9   | I_O     |      1 |ARRAY of INTEGER         |      10  |
|   19.6   | I       |      1 |ARRAY of INTEGER         |     100  |
|   18.6   | O       |      1 |ARRAY of INTEGER         |     100  |
|   18.6   | I_O     |      1 |ARRAY of INTEGER         |     100  |
|   19.1   | I       |      1 |ARRAY of INTEGER         |   10000  |
|   17.7   | O       |      1 |ARRAY of INTEGER         |   10000  |
|   19.4   | I_O     |      1 |ARRAY of INTEGER         |   10000  |
|   18.7   | I       |      1 |RECORD of INTEGER        |       1  |
|   19.2   | O       |      1 |RECORD of INTEGER        |       1  |
|   19.4   | I_O     |      1 |RECORD of INTEGER        |       1  |
|   17.4   | I       |      1 |RECORD of INTEGER        |     100  |
|   19.8   | O       |      1 |RECORD of INTEGER        |     100  |
|   17.8   | I_O     |      1 |RECORD of INTEGER        |     100  |
|   21.2   | I       |      1 |UNCONSTRAINED ARRAY      |       1  |
|   21.1   | O       |      1 |UNCONSTRAINED ARRAY      |       1  |
|   21.2   | I_O     |      1 |UNCONSTRAINED ARRAY      |       1  |
|   21.2   | I       |      1 |UNCONSTRAINED ARRAY      |     100  |
|   21.1   | O       |      1 |UNCONSTRAINED ARRAY      |     100  |
|   21.2   | I_O     |      1 |UNCONSTRAINED ARRAY      |     100  |
|   21.1   | I       |      1 |UNCONSTRAINED ARRAY      |   10000  |
|   21.1   | O       |      1 |UNCONSTRAINED ARRAY      |   10000  |
|   21.1   | I_O     |      1 |UNCONSTRAINED ARRAY      |   10000  |
|   19.3   | I       |      1 |UNCONSTRAINED RECORD     |       1  |
|   19.2   | O       |      1 |UNCONSTRAINED RECORD     |       1  |
|   20.5   | I_O     |      1 |UNCONSTRAINED RECORD     |       1  |
|   19.3   | I       |      1 |UNCONSTRAINED RECORD     |     100  |
|   19.1   | O       |      1 |UNCONSTRAINED RECORD     |     100  |
|   20.4   | I_O     |      1 |UNCONSTRAINED RECORD     |     100  |
```

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

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
0.4		0		
1.4	I	1	INTEGER	
2.6	O	1	INTEGER	
2.2	I_O	1	INTEGER	
15.2	I	10	INTEGER	
29.3	O	10	INTEGER	
30.4	I_O	10	INTEGER	
152.8	I	100	INTEGER	
315.0	O	100	INTEGER	
381.4	I_O	100	INTEGER	
1.6	I	1	ENUMERATION	
2.5	O	1	ENUMERATION	
2.5	I_O	1	ENUMERATION	
15.1	I	10	ENUMERATION	
30.0	O	10	ENUMERATION	
30.0	I_O	10	ENUMERATION	
153.2	I	100	ENUMERATION	
311.6	O	100	ENUMERATION	
353.6	I_O	100	ENUMERATION	
-0.1	I	1	ARRAY of INTEGER	1
1.0	O	1	ARRAY of INTEGER	1
0.0	I_O	1	ARRAY of INTEGER	1
0.3	I	1	ARRAY of INTEGER	10
0.5	O	1	ARRAY of INTEGER	10
0.5	I_O	1	ARRAY of INTEGER	10
0.0	I	1	ARRAY of INTEGER	100
0.4	O	1	ARRAY of INTEGER	100
0.4	I_O	1	ARRAY of INTEGER	100
1.1	I	1	ARRAY of INTEGER	10000
0.9	O	1	ARRAY of INTEGER	10000
0.5	I_O	1	ARRAY of INTEGER	10000
0.4	I	1	RECORD of INTEGER	1
0.0	O	1	RECORD of INTEGER	1
0.2	I_O	1	RECORD of INTEGER	1
0.4	I	1	RECORD of INTEGER	100
0.6	O	1	RECORD of INTEGER	100
0.4	I_O	1	RECORD of INTEGER	100
2.5	I	1	UNCONSTRAINED ARRAY	1
2.5	O	1	UNCONSTRAINED ARRAY	1
2.5	I_O	1	UNCONSTRAINED ARRAY	1
2.6	I	1	UNCONSTRAINED ARRAY	100
2.6	O	1	UNCONSTRAINED ARRAY	100
2.5	I_O	1	UNCONSTRAINED ARRAY	100
2.1	I	1	UNCONSTRAINED ARRAY	10000
2.6	O	1	UNCONSTRAINED ARRAY	10000
1.1	I_O	1	UNCONSTRAINED ARRAY	10000
2.7	I	1	UNCONSTRAINED RECORD	1
2.8	O	1	UNCONSTRAINED RECORD	1
2.7	I_O	1	UNCONSTRAINED RECORD	1
2.9	I	1	UNCONSTRAINED RECORD	100
2.7	O	1	UNCONSTRAINED RECORD	100
2.7	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
18.5		0		
21.6	I	1	INTEGER	
23.2	O	1	INTEGER	
24.9	I_O	1	INTEGER	
34.4	I	10	INTEGER	
48.9	O	10	INTEGER	
49.1	I_O	10	INTEGER	
173.4	I	100	INTEGER	
344.6	O	100	INTEGER	
333.5	I_O	100	INTEGER	
21.4	I	1	ENUMERATION	
24.6	O	1	ENUMERATION	
24.6	I_O	1	ENUMERATION	
34.4	I	10	ENUMERATION	
50.0	O	10	ENUMERATION	
51.0	I_O	10	ENUMERATION	
173.1	I	100	ENUMERATION	
374.7	O	100	ENUMERATION	
359.4	I_O	100	ENUMERATION	
23.1	I	1	ARRAY of INTEGER	1
23.3	O	1	ARRAY of INTEGER	1
23.0	I_O	1	ARRAY of INTEGER	1
22.5	I	1	ARRAY of INTEGER	10
21.2	O	1	ARRAY of INTEGER	10
23.9	I_O	1	ARRAY of INTEGER	10
26.3	I	1	ARRAY of INTEGER	100
22.2	O	1	ARRAY of INTEGER	100
21.7	I_O	1	ARRAY of INTEGER	100
23.4	I	1	ARRAY of INTEGER	10000
22.0	O	1	ARRAY of INTEGER	10000
21.5	I_O	1	ARRAY of INTEGER	10000
22.9	I	1	RECORD of INTEGER	1
23.8	O	1	RECORD of INTEGER	1
23.1	I_O	1	RECORD of INTEGER	1
23.0	I	1	RECORD of INTEGER	100
22.3	O	1	RECORD of INTEGER	100
21.7	I_O	1	RECORD of INTEGER	100

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

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
17.6		0		
19.7	I	1	INTEGER	
20.2	O	1	INTEGER	
19.9	I_O	1	INTEGER	
31.9	I	10	INTEGER	
46.3	O	10	INTEGER	
46.1	I_O	10	INTEGER	
170.0	I	100	INTEGER	
321.8	O	100	INTEGER	
322.0	I_O	100	INTEGER	
19.6	I	1	ENUMERATION	
20.4	O	1	ENUMERATION	
20.2	I_O	1	ENUMERATION	
33.4	I	10	ENUMERATION	
46.4	O	10	ENUMERATION	
46.4	I_O	10	ENUMERATION	
169.8	I	100	ENUMERATION	
324.5	O	100	ENUMERATION	
321.1	I_O	100	ENUMERATION	
19.7	I	1	ARRAY of INTEGER	1
20.7	O	1	ARRAY of INTEGER	1
20.7	I_O	1	ARRAY of INTEGER	1
19.6	I	1	ARRAY of INTEGER	10
19.1	O	1	ARRAY of INTEGER	10
19.1	I_O	1	ARRAY of INTEGER	10
19.6	I	1	ARRAY of INTEGER	100
19.1	O	1	ARRAY of INTEGER	100
19.1	I_O	1	ARRAY of INTEGER	100
19.7	I	1	ARRAY of INTEGER	10000
19.1	O	1	ARRAY of INTEGER	10000
19.2	I_O	1	ARRAY of INTEGER	10000
19.9	I	1	RECORD of INTEGER	1
20.5	O	1	RECORD of INTEGER	1
19.9	I_O	1	RECORD of INTEGER	1
19.8	I	1	RECORD of INTEGER	100
19.1	O	1	RECORD of INTEGER	100
18.7	I_O	1	RECORD of INTEGER	100
21.3	I	1	UNCONSTRAINED ARRAY	1
20.7	O	1	UNCONSTRAINED ARRAY	1
20.7	I_O	1	UNCONSTRAINED ARRAY	1
21.2	I	1	UNCONSTRAINED ARRAY	100
20.8	O	1	UNCONSTRAINED ARRAY	100
20.8	I_O	1	UNCONSTRAINED ARRAY	100
21.3	I	1	UNCONSTRAINED ARRAY	10000
20.8	O	1	UNCONSTRAINED ARRAY	10000
20.8	I_O	1	UNCONSTRAINED ARRAY	10000
20.3	I	1	UNCONSTRAINED RECORD	1
22.9	O	1	UNCONSTRAINED RECORD	1
21.0	I_O	1	UNCONSTRAINED RECORD	1
20.3	I	1	UNCONSTRAINED RECORD	100
21.0	O	1	UNCONSTRAINED RECORD	100
21.0	I_O	1	UNCONSTRAINED RECORD	100

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

Time	Direction	# Passed	Type	Size of
(microsec.)	Passed	in Call	Passed	Passed Var
22.2		0		
27.2	I	1	INTEGER	
27.8	O	1	INTEGER	
27.5	I_O	1	INTEGER	
39.3	I	10	INTEGER	
53.6	O	10	INTEGER	
53.4	I_O	10	INTEGER	
180.1	I	100	INTEGER	
328.6	O	100	INTEGER	
330.3	I_O	100	INTEGER	
26.5	I	1	ENUMERATION	
26.9	O	1	ENUMERATION	
27.0	I_O	1	ENUMERATION	
38.9	I	10	ENUMERATION	
54.3	O	10	ENUMERATION	
53.4	I_O	10	ENUMERATION	
176.1	I	100	ENUMERATION	
331.7	O	100	ENUMERATION	
337.3	I_O	100	ENUMERATION	
26.8	I	1	ARRAY of INTEGER	1
27.7	O	1	ARRAY of INTEGER	1
27.6	I_O	1	ARRAY of INTEGER	1
27.2	I	1	ARRAY of INTEGER	10
26.2	O	1	ARRAY of INTEGER	10
25.8	I_O	1	ARRAY of INTEGER	10
27.1	I	1	ARRAY of INTEGER	100
26.3	O	1	ARRAY of INTEGER	100
26.2	I_O	1	ARRAY of INTEGER	100
31.6	I	1	ARRAY of INTEGER	10000
27.9	O	1	ARRAY of INTEGER	10000
27.9	I_O	1	ARRAY of INTEGER	10000
27.2	I	1	RECORD of INTEGER	1
27.7	O	1	RECORD of INTEGER	1
26.7	I_O	1	RECORD of INTEGER	1
27.2	I	1	RECORD of INTEGER	100
26.2	O	1	RECORD of INTEGER	100
26.2	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
0.0	1	Integer	
-1.3	10	Integer	
27.5(1)	100	Integer	
0.0	1	String	1
0.0	1	String	10
0.1	1	String	100
-0.5	1	Enumeration	
-1.8	10	Enumeration	
-1.5	100	Enumeration	
0.1	1	Integer Array	1
1.0	1	Integer Array	10
1.3	1	Integer Array	100
-0.4	1	Integer Array	1000
143.0	1	1-D Dynamically bounded Array	1
779.0	1	1-D Dynamically bounded Array	10
148.9	1	2-D Dynamically bounded Array	1
160.6	1	3-D Dynamically bounded Array	1
-1.0	1	Record of Integer	1
-0.6	1	Record of Integer	10
-0.8	1	Record of Integer	100

Note: Times reported include any deallocation required upon leaving the sc

(1) - Other runs hace indicated that this value is too small to measure.
of the declared variables.

Number of Iterations = 1000 * 10

Dynamic Allocation with NEW allocator

Time (microsec.)	# Declared	Type Declared	Size of Object
227.0	1	Integer	1
230.0	1	Enumeration	1
239.9	1	Record of Integer	1
260.1	1	Record of Integer	5
270.0	1	Record of Integer	10
270.0	1	Record of Integer	20
309.9	1	Record of Integer	50
340.0	1	Record of Integer	100
220.0	1	String	1
229.9	1	String	10
239.9	1	String	100
220.0	1	Integer Array	1
260.1	1	Integer Array	10
249.9	1	Integer Array	100
270.0	1	Integer Array	1000
200.0	1	1-D Dynamically Bounded Array	1
260.0	1	1-D Dynamically Bounded Array	10
280.0	1	2-D Dynamically Bounded Array	1
1140.1	1	2-D Dynamically Bounded Array	100
300.0	1	3-D Dynamically Bounded Array	1
3370.1	1	3-D Dynamically Bounded Array	1000

Note: The times reported include the time required to deallocate objects. This version of the compiler does deallocate memory when UNCHECKED_DEALLOCATION is called.

```
Number of Iterations = 1000 * 10


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

      Exception raised and handled in a block
         0.0 uSEC.     User Defined, Not Raised
       315.0 uSEC.     User Defined
       343.0 uSEC.     Constraint Error, Implicitly Raised
       325.0 uSEC.     Constraint Error, Explicitly Raised
          (1)          Numeric Error, Implicitly Raised
       361.0 uSEC.     Numeric Error, Explicitly Raised
       372.0 uSEC.     Tasking Error, Explicitly Raised

      Exception raised in a procedure and handled in the calling unit
         1.0 uSEC.     User Defined, Not Raised
       544.0 uSEC.     User Defined
       574.0 uSEC.     Constraint Error, Implicitly Raised
       555.0 uSEC.     Constraint Error, Explicitly Raised
      2551.0 uSEC.     Numeric Error, Implicitly Raised
       599.0 uSEC.     Numeric Error, Explicitly Raised
       613.0 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:    20.4 milliseconds.
----------------------------------------------------------------
```

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

For test number          1
Task elaborate, activate, terminate time:    20.5 milliseconds.
----------------------------------------------------------------
```

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

For test number          1
Task elaborate, activate, terminate time:    17.5 milliseconds.
----------------------------------------------------------------
```

```
Rendezvous Time:   No Parameters Passed
Number of Iterations = 100
-------------------------------------------------------------
Task Rendezvous Time:    3.4 milliseconds
-------------------------------------------------------------
```

```
Number of Iterations = 10000


Clock function calling overhead : 3486.00  microseconds
Clock function calling overhead : 3554.00  microseconds
Clock function calling overhead : 3586.00  microseconds
Clock function calling overhead : 3599.00  microseconds
Clock function calling overhead : 3608.00  microseconds
Clock function calling overhead : 3560.00  microseconds
Clock function calling overhead : 3565.00  microseconds
Clock function calling overhead : 3609.00  microseconds
```

```
Number of Iterations = 10000 * 10


          TIME and DURATION math
          ======================
Microseconds   Operation
  716.00        Time := var_time + var_duration
  750.90       Time := Var_time - Var_duration
  774.00       Time := Var_duration + Var_time
  756.10       Time := Var_time + Const_duration
  812.50       Time := Var_time - Const_duration
  779.30       Time := Const_duration + Var_time
   49.60       Duration := Var_time - Var_time
    6.30       Duration := var_duration + var_duration
    9.00       Duration := Var_duration - Var_duration
    6.80       Duration := Var_duration + Const_duration
    6.30       Duration := Var_duration - Const_duration
    6.30       Duration := Const_duration + Var_duration
    7.30       Duration := Const_duration - Var_duration
    1.10       Duration := Const_duration + Const_duration
    1.60       Duration := Const_duration - Const_duration
```

The following results are consistent with the analysis given in the report. 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. 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 requested by the LRM.

```
Delay Statement Test
Number of Iterations = 1
--------------------------------------------
For case number            1
Desired delay time:      0.001037597656250  seconds.
Actual delay time:       0.939941406250000  seconds.

For case number            2
Desired delay time:      0.002075195312500  seconds.
Actual delay time:       0.659973144531250  seconds.

For case number            3
Desired delay time:      0.003112792968750  seconds.
Actual delay time:       0.679992675781250  seconds.

For case number            4
Desired delay time:      0.004150390625000  seconds.
Actual delay time:       0.059997558593750  seconds.

For case number            5
Desired delay time:      0.005187988281250  seconds.
Actual delay time:       0.559936523437500  seconds.

For case number            6
Desired delay time:      0.006225585937500  seconds.
Actual delay time:       0.639953613281250  seconds.

For case number            7
Desired delay time:      0.007263183593750  seconds.
Actual delay time:       0.779968261718750  seconds.

For case number            8
Desired delay time:      0.008300781250000  seconds.
Actual delay time:       0.869934082031250  seconds.

For case number            9
Desired delay time:      0.009338378906250  seconds.
Actual delay time:       0.659973144531250  seconds.

For case number            10
Desired delay time:      0.010375976562500  seconds.
Actual delay time:       0.409973144531250  seconds.

For case number            11
Desired delay time:      0.011413574218750  seconds.
Actual delay time:       0.539978027343750  seconds.

For case number            12
Desired delay time:      0.012451171875000  seconds.
Actual delay time:       0.489990234375000  seconds.

For case number            13
Desired delay time:      0.013488769531250  seconds.
Actual delay time:       0.709960937500000  seconds.

For case number            14
Desired delay time:      0.014526367187500  seconds.
Actual delay time:       0.789978027343750  seconds.
```

```
Delay Statement Test
Number of Iterations = 1
-------------------------------------------------
For case number            1
Desired delay time:      0.097656250000000 seconds.
Actual delay time:       0.940002441406250 seconds.

For case number            2
Desired delay time:      0.195312500000000 seconds.
Actual delay time:       0.660095214843750 seconds.

For case number            3
Desired delay time:      0.292968750000000 seconds.
Actual delay time:       0.690063476562500 seconds.

For case number            4
Desired delay time:      0.390625000000000 seconds.
Actual delay time:       1.210083007812500 seconds.

For case number            5
Desired delay time:      0.488281250000000 seconds.
Actual delay time:       1.300048828125000 seconds.

For case number            6
Desired delay time:      0.585937500000000 seconds.
Actual delay time:       0.640075683593750 seconds.

For case number            7
Desired delay time:      0.683593750000000 seconds.
Actual delay time:       0.790100097656250 seconds.

For case number            8
Desired delay time:      0.781250000000000 seconds.
Actual delay time:       0.880004882812500 seconds.

For case number            9
Desired delay time:      0.878906250000000 seconds.
Actual delay time:       1.660095214843750 seconds.

For case number           10
Desired delay time:      0.976562500000000 seconds.
Actual delay time:       1.420104980468750 seconds.

For case number           11
Desired delay time:      1.074218750000000 seconds.
Actual delay time:       1.560058593750000 seconds.

For case number           12
Desired delay time:      1.171875000000000 seconds.
Actual delay time:       1.500061035156250 seconds.

For case number           13
Desired delay time:      1.269531250000000 seconds.
Actual delay time:       1.720092773437500 seconds.

For case number           14
Desired delay time:      1.367187500000000 seconds.
Actual delay time:       1.810058593750000 seconds.

For case number           15
Desired delay time:      1.464843750000000 seconds.
Actual delay time:       1.530090332031250 seconds.

For case number           16
Desired delay time:      1.562500000000000 seconds.
Actual delay time:       1.610046386718750 seconds.
```

```
Delay Statement Test
Number of Iterations = 1
-----------------------------------------------------
For case number              1
Desired delay time:        0.213623046875000 seconds.
Actual delay time:         0.950012207031250 seconds.

For case number              2
Desired delay time:        0.427246093750000 seconds.
Actual delay time:         0.679992675781250 seconds.

For case number              3
Desired delay time:        0.640869140625000 seconds.
Actual delay time:         0.700012207031250 seconds.

For case number              4
Desired delay time:        0.854492187500000 seconds.
Actual delay time:         1.229980468750000 seconds.

For case number              5
Desired delay time:        1.068115234375000 seconds.
Actual delay time:         1.299987792968750 seconds.

For case number              6
Desired delay time:        1.281738281250000 seconds.
Actual delay time:         1.650024414062500 seconds.

For case number              7
Desired delay time:        1.495361328125000 seconds.
Actual delay time:         1.799987792968750 seconds.

For case number              8
Desired delay time:        1.708984375000000 seconds.
Actual delay time:         1.880004882812500 seconds.

For case number              9
Desired delay time:        1.922607421875000 seconds.
Actual delay time:         2.669982910156250 seconds.

For case number              10
Desired delay time:        2.136230468750000 seconds.
Actual delay time:         2.429992675781250 seconds.

For case number              11
Desired delay time:        2.349853515625000 seconds.
Actual delay time:         2.559997558593750 seconds.

For case number              12
Desired delay time:        2.563476562500000 seconds.
Actual delay time:         3.510009765625000 seconds.

For case number              13
Desired delay time:        2.777099609375000 seconds.
Actual delay time:         2.719970703125000 seconds.

For case number              14
Desired delay time:        2.990722656250000 seconds.
Actual delay time:         3.809997558593750 seconds.

For case number              15
Desired delay time:        3.204345703125000 seconds.
Actual delay time:         3.539978027343750 seconds.

For case number              16
Desired delay time:        3.417968750000000 seconds.
Actual delay time:         4.229980468750000 seconds.
```