

# Chapter 3

## The Time Problem

Russell M. Clapp    Trevor Mudge

The University of Michigan

The notion of time in Ada is necessary since the language is intended for use in programming embedded real-time systems. In this chapter, we examine time related issues in Ada and their relation to our approach to benchmarking. There are two aspects of the time problem area that must be considered in performance evaluation. The first is a need for benchmark programs to evaluate the timing related features of the language. Secondly, the existence of timing support provides the basis for developing a method within the language for measuring benchmark execution times. In order to develop an accurate and reliable method for timing measurement, the support for timing provided by the language must be calibrated using benchmark programs. We discuss these two inter-related areas of the time problem in the paragraphs below, after first reviewing the notion of time present in Ada.

### 3.1 Ada Time Units

There are several entities in Ada that relate to time. The following list of entities is reproduced from [10]:

- the data type `TIME`, objects of which type are used to hold an internal representation of an absolute point in time;
- the data type `DURATION`, objects of which type are used to hold values for intervals of time;
- the value `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;
- the value `SYSTEM'TICK`, which is defined as the basic system unit of time;
- a predefined package, `CALENDAR`, which provides functions to perform arithmetic on objects of type `TIME` or `DURATION`;

- a predefined function, `CLOCK`, which returns a value of type `TIME` corresponding to the current time;
- the operation `delay` which allows a task to suspend itself for a period of time.

The first two items are predefined data types for representing either points in time or intervals of time. A point in time has several components associated with it that represent various levels of precision in a time value, e.g., year, month, day, hour, minute, second, microsecond, etc. The type `TIME` is private, so the representation used should be hidden from the user. The multiple levels of precision suggest, though, that the representation of a point in time may be more complex than a single word value. Because of this, the overhead involved in computations involving values of type `TIME` will be different and probably greater than in the case of other predefined data types.

The representation for an interval of time is simpler. `DURATION` is a predefined fixed point type whose values are interpreted to be in seconds. Because it is a fixed point type, the values of objects of this type are multiples of the value `DURATION'SMALL`. This value, then, specifies the amount of the precision possible in representing an interval of time. It is important to note, however, that the value of `DURATION'SMALL` in no way implies performance of a system for time measurements or scheduling. The clock resolution is determined ultimately by the hardware. `DURATION'SMALL` is chosen based on word size and convenience of representing a wide range of interval values with a reasonable amount of precision.

A similar statement can be made regarding the timing environment available and the value `SYSTEM.TICK`. The phrase “basic system unit of time” is not very specific, and one might think it refers to one of several possibilities, e.g., time-slice interval, minimum delay value, CPU clock cycle, or hardware clock resolution. Several compilers do use this value to indicate the resolution of the `CLOCK` function, but this is not always the case. It is best to determine this resolution experimentally, as described in later sections.

The `CALENDAR` package contains subprograms to perform arithmetic on objects of types `TIME` and `DURATION` as well as extract date components from an object of type `TIME` or combine such components into a single `TIME` object. The `CALENDAR` package also contains the `CLOCK` function. The `CLOCK` function returns a value based on some underlying hardware clock or timer. The increment in time to the clock or timer is the resolution time of the system. If this increment is the same as the tick for the `CLOCK` function, it is also the `CLOCK` resolution. If the execution overhead for evaluating the `CLOCK` function is less than its resolution, then successive calls to the `CLOCK` function may return the same result.

The `delay` operation allows scheduling and synchronization of tasks to be influenced by time. Delays are specified with a time interval of type `DURATION`. A delay may suspend a task for an amount of time no less than that specified, or the delay may be used in conjunction with an entry call or one or more accept statements. In the case of rendezvous related delays, the delay specifies a bound on the time a calling or accepting task is allowed to wait for a specified rendezvous to begin. In any case, there are several possibilities for the implementation of the delay operation. The LRM places no limits on the length of a delay, except that it last at least as long as the time interval specified. This situation gives an implementation a lot of freedom, and the particular implementation strategy used can greatly affect a program's execution.

## 3.2 Timing Techniques

There are many factors which come into play when determining the method to be used for timing overhead and latencies in benchmark programs. The most important issue to address is the structuring of the benchmark code. It is necessary to ensure that the benchmark measures exactly what it is designed to measure. It is also important that the timing environment be well understood, so that the desired accuracy and precision in the measurement can be obtained. Additionally, careful examination of the results must be made, so that undesired effects from the hardware, operating system, or run-time system can be discounted. We discuss each of these areas in detail below.

### 3.2.1 Feature Isolation

In order to measure a latency time or overhead for a specific language feature or set of statements, it is necessary to subtract out any overhead for control statements that aid in performing the measurement. In order to do this, the overhead of the control statements must be measured separately. This is done by textually repeating the code for the test but omitting the feature or set of statements that are the target of the benchmark. Because the resolution of the timer is usually greater than the time being measured, it is common for the benchmark code to include loops around the features being measured so that they are repeated a large number of times. The measured value is then divided by the number of iterations. The form of the overall benchmark, then, is made up of two loops: the test loop and the control loop. This benchmarking technique has been referred to as the “dual loop” approach [3].

Although this technique was designed for measuring language feature overhead, it can also be used to measure running times for synthetic benchmarks and applications. The feature statement is simply replaced by the statements that make up the application being measured. This technique may not be necessary if the running time of the synthetic benchmark or application is long in comparison to the CLOCK resolution. If this is not the case, the dual loop approach provides a way to obtain more accuracy and precision in the measurement through the use of repetition with timed feature isolation.

In order for dual loop coding approach to be successful, it is important to ensure that the measurements it produces are valid. The major factor to consider when designing the dual loop code is the effect of compiler optimizations. Although code optimizations are generally welcomed as a way to improve performance, they can also invalidate a measurement made by using the dual loop approach. Techniques must be incorporated to prevent an optimizing compiler from changing the program in any way that invalidates a measurement. The code generated for the test loop and control loop must not be different, except for the quantity being measured. Of course, any optimizations to the actual feature being measured welcomed and encouraged. Chapter 6 discusses possible compiler optimizations. Chapter 7 provides examples of the dual loop code structure used in the PIWG benchmarking suite.

### 3.2.2 Timing Accuracy and Precision

In addition to correctly structuring the code for the benchmark tests, it is important to evaluate the timing mechanisms available and use them properly. The accuracy and precision of any clocks used must be measured, since these values determine the running time of the benchmark, which in turn determines the precision and accuracy of the results due to the measurement technique.

### 3.2.2.1 Clocks

There are several options possible when choosing a timing mechanism for use in benchmark programs. The most obvious choice in this case is the Ada CLOCK function of the CALENDAR package. The CLOCK function measures real time (also referred to as “wall clock time”) by definition of the language. Using the CLOCK function for timing in benchmarks has both advantages and disadvantages. The main advantage is portability, since the time intervals may be determined by taking the difference of two CLOCK readings in an implementation independent manner. The main disadvantage is that, since CLOCK returns real time, the benchmark must be run in a situation where it has sole access to all available CPU cycles. If any intervening processes are present, e.g., in a time-shared system, they will steal away CPU cycles from the benchmark while the CLOCK is still ticking. This situation would provide an inaccurate measure of the benchmark’s running time.

Another possibility for timing in benchmark programs is to use a CPU or “virtual” timer. A timer of this type measures the time actually used by the program. It does not tick when the program is blocked and another job is executing. This type of timer is generally provided in time-shared systems (e.g. Unix and VMS). There are two main disadvantages in using CPU timers. The first is portability. Although the timer may be invoked by calling a language level function, that function must be rewritten for each operating system that the benchmark programs are run with. The second disadvantage is the degree of uncertainty present when using CPU timers. In the presence of other jobs, CPU timers charge ticks to the running process when the wall clock is updated. Since context switches can occur at any time, it is possible for time to be charged to the active processes inaccurately. A few informal tests have been done at Michigan that demonstrates variability of CPU time readings in the presence of additional system load. While the timers may be accurate enough in the long run for accounting purposes, a confidence level for short time periods must be established before using them in conjunction with benchmark programs.

Yet another option for timing in benchmark programs is to use machine dependent hardware timers. These timers can generally be read directly without operating system interference, and they usually measure real time. When they are present, hardware timers usually provide a finer resolution than timers available through the operating system. These timers are critical in the case of embedded systems where there is no operating system. While lower reading overhead and higher resolution are desirable qualities for a timer, the problem of portability still remains in this case.

In the case of embedded systems, clock resolution and accuracy are even more important. In order to maximize these features in such a system, logic analyzers and other high resolution external hardware timers can be used. While providing the highest amount of accuracy and precision possible, these timers often require knowledge of the assembly code generated by the benchmark so that a trigger for the external timer can be determined. This type of timing is not at all machine independent, and cannot be expressed at the language level.

The PIWG suite currently uses both the Ada CLOCK function and an operating system dependent CPU timer for measuring time intervals. Because the CLOCK function measures real time and the CPU time may vary with system load, it is required that the benchmark program be the only active program in the system. When available or appropriate, the definition of the system dependent clock function can be changed to take advantage of additional accuracy and precision, as described above. The use of multiple clocks provides some additional confidence in the results when they agree, and can help pinpoint problems when they do not.

### 3.2.2.2 Clock Resolution

Independent of which type of clock is used to measure time intervals, its precision must be determined since this affects the benchmark's running time and the precision of the results. As stated in Chap. 2, the effective precision of a clock is the resolution, or interval of time by which the clock is incremented at each tick. This value can be determined using the second differencing technique explained in detail in [11]. We outline the basic approach below.

The idea behind second differencing is that, independent of the time needed to read the clock and its effective precision, the resolution can be determined by a series of nearly consecutive, equally spaced clock readings. By equally spaced, we mean clock readings with the same set of other instructions being executed between each read. If we assume that the resolution can be exactly represented by a model number that is an integer multiple of DURATION'SMALL, then we can determine the clock resolution by taking the second difference of the string of values produced from the consecutive clock readings.

A sequence of first differences is produced by computing the difference between all adjacent values in the original sequence of clock readings. Applying this differencing operation again to the sequence of first differences produces a sequence of second differences. As stated in [10], the values present in the second difference sequence are one of  $\tau$ ,  $-\tau$ , or 0, where  $\tau$  is the clock resolution. Furthermore, as stated in [10], the number of zeros present in the second difference sequence can be bounded and reduced by adding instructions between consecutive readings of the clock.

If we drop the assumption that the resolution may be exactly represented by a model number for type DURATION, the process of determining the clock resolution becomes more involved. Depending on the implementation of the CLOCK function or some other clock routine and its Ada interface, it is possible for first or second difference values to be equal to DURATION'SMALL when they should be equal to 0. In this situation, the larger of any values present in a second difference string should be regarded as the clock resolution. This problem arises because a value that should be equal to the clock resolution can instead only approximate it. The difference between this value and the actual clock resolution is less than DURATION'SMALL. Thus, two distinct values that differ by DURATION'SMALL could both be representations of the clock resolution. This problem is addressed in more detail in the paper by Pollack and Campbell that appears in Chap. 7.

Another problem that has occurred with this technique of clock calibration is the case where the real time clock of a system is artificially incremented by a value much smaller than the resolution between consecutive clock readings. This change in the value of the clock is not due to a tick, but instead occurs because the operating system wants to guarantee that no two clock readings will return the same result. This property is useful when there is a need to dynamically generate unique identifiers, but it confuses the semantics of the clock.

This style of real time clock implementation has been observed at Michigan and by others [36] in the Sun-3 and Sun-4 series computers. The clock is incremented by one microsecond between readings unless enough time has passed to tick the clock to the next even multiple of 20 milliseconds. When calibrating clocks of this type, closer scrutiny is required when examining the original readings, the first difference sequence, and second difference sequences.

### 3.2.2.3 Accuracy

Clock accuracy is important to gauge as it determines the degree of reliability of benchmark results. It is also desirable to measure the accuracy of the CLOCK function since it is a performance parameter of the compiler and run-time system. One way to measure accuracy is to compare clock readings with values from another clock. At the language and system level, this would involve comparing multiple consecutive readings of all clocks available, e.g., CLOCK, a CPU clock, a hardware clock, or an external timer. Since all of the internal clocks may rely on the same hardware timer, we would expect them to agree for the most part. Any discrepancies may expose differences in the implementation or varying amounts of overhead in reading the clock. Ultimately, in order to verify the validity of the timing hardware, the updating of the clock value must be compared against an external standard.

Another issue that is important to the real-time programmer and which may affect accuracy is the overhead encountered in reading the clock. This is mainly a concern of the CLOCK function provided by the language. Measuring overhead and comparing it to the resolution will give an indication of how “stale” the CLOCK reading may be. Also, if a programmer wishes to use the CLOCK function as part of a calculation to determine a value for a **delay** statement, the overhead involved in calling CLOCK could be critical.

However, the overhead involved in reading the clock does not theoretically affect the second differencing technique described above. A longer overhead has an effect analogous to inserting other instructions between clock readings. If the clock overhead is greater than the resolution, though, it is necessary to compute the second differences to determine the clock resolution. If the overhead is less than the resolution, computing the first differences is sufficient to produce the resolution.

As for measuring time intervals for benchmarks, the clock overhead will not greatly affect the results as long as it is constant. Intuitively, this results from the notion that the overhead will be split into two pieces, the part before the actual instruction that reads a time register and the part that occurs after. Two readings of the clock then measure the running time of the instructions executing between the calls as well as one after part and one before part of the clock reading overhead. Therefore, a timed interval contains the total time needed for one reading of the clock. If the overhead is on the order of the clock resolution, its effect on the measurement will be diminished by the number of times the measured feature was repeated. If the overhead is much greater than the resolution, its value can be subtracted from each time interval measured.

### 3.2.2.4 Measurement Precision and Accuracy

The precision of the time measurements is determined by the resolution of the clock used and the number of times the feature being measured in the loop was repeated. Because loops are used, we call this repetition count the number of *iterations*. As reported in [10], the time a loop takes to execute and the time measured for it using a clock can differ by as much as the resolution of the clock. When this value is divided by the number of iterations (call it  $N$ ), we find that the measured value for a single execution of a feature is within a value  $\tau/N$  of the actual time (where  $\tau$  is the clock resolution). Assuming no other effects are present that invalidate the measurement, we can say that the measurement is to a precision of  $\tau/N$ , and the accuracy of the reported value is within  $\pm\tau/N$  of the actual value. Since  $\tau/N$  indicates a bound on the amount that the measured value can vary

from the actual, a percentage of certainty can be calculated. If it is desired that the reported results be within 1% of the actual, one only need guarantee that  $\tau/N$  be less than 1% of the measured time. This can be done by increasing the number of iterations until the condition is met.

### 3.2.2.5 Measurement Stability

Because of several different effects that interfere with benchmark measurements (as described below), we would like to specify some stability criteria to help increase confidence in a measurement. Since the clock resolution can be determined and the number of iterations is under programmer control, the amount of error introduced by the repetition technique alone can be minimized. As stated above, we can limit this particular source of error to 1% or even less.

To reach this goal of 1% error, the number of iterations for a test is increased until the difference in times for the test loop and the control loop is greater than  $100 * \tau$ . If we assume no other effects are present that may skew the results, the value determined for the measurement will be within one clock tick of the actual time used.  $\pm\tau$  compared to  $100 * \tau$  is an error of  $\pm 1\%$ . The value reported for a single iteration is also within 1%, the precision of this value is  $\tau/N$  as stated above.

In situations where the clock resolution is coarse and the feature being measured requires very little time to execute, the number of iterations may be prohibitively large. Allowances for additional error must be made in these cases and considered when analyzing the results. Also, it is important to note that other effects may be present that introduce additional error into the results. These effects are described in the next section.

## 3.2.3 Problems with Dual Loop Approach

There are several problems that can occur in different compiler implementations running on different computer systems that can distort the benchmark results when using the measurement techniques described above. We classify the major problems into three categories:

1. Translation Anomalies
2. Code Placement Effects
3. Operating and Run-Time System Interference

Although 3. was discussed in [10], points 1. and 2. have been identified by users of the PIWG and Michigan suites since 1986. All three of these problems have been discussed in detail in [2]. They were also discussed in April 1988 at a PIWG workshop [34].

### 3.2.3.1 Translation Anomalies

A translation anomaly occurs when the compiler generates correct code that differs from that expected. The principle causes of translation anomalies result from a failure in the technique used for blocking an optimizer or from a compiler that generates different machine code for two pieces of textually identical Ada code. The first case occurs when improper techniques for optimization blocking are used.

A concerted effort must be made at design time to ensure that all optimization blocking techniques cannot be circumvented. Prototyping the benchmark on several systems can also aid in detecting loopholes that optimizers might find, but this should not be used exclusively.

The second case of translation anomaly can occur if a compiler generates machine code based on some global conditions in addition to the original Ada source. For example, a subprogram's entry code may be longer or shorter if it is the first one declared in a package. Other examples include subprogram calls being long or short based on their relative position to the caller or NOP instructions being added in some cases to provide word alignment. These problems can be detected by repeating the benchmark loops several times textually and measuring times for all the control and test loops separately. If more information is known about why any discrepancies may exist (e.g. compiler generated assembly code is available), some determination may be made as to which loop times should be considered valid. This code repetition technique can also detect other anomalies as we shall see next.

### 3.2.3.2 Code Placement Effects

Additional timing anomalies can occur when running benchmarks due to the effect of code placement in memory. Experiments have shown that textually identical loops may have different running times if code for a loop crosses a page boundary (even if there is no page fault) or is aligned on a halfword boundary [2].

Other problems arise if the benchmark code cannot entirely fit inside a cache (if present) or inside of main memory. Cache misses and/or page faults invoke system activity and increase execution overhead. If these effects are not equally spread across all timing loops, the results will be distorted. In [10], the recommendation was made that all benchmarks be made small enough to avoid the effects of paging. In some cases, this requires a benchmark to "undo" certain actions previously done, so that large amounts of memory are not consumed [7]. When caches are present, it would be preferable if the benchmark code could reside entirely in the cache. A run through the code before timing begins can bring the code into (or "warm up") the cache and main memory.

Testing with multiple textually identical loops can aid in detecting any of these conditions. The benchmarks should also be repeated textually within the code with multiple complete runs made to determine if any of these conditions are significantly affecting the results.

### 3.2.3.3 Operating and Run-Time System Interference

Interference from the operating system and/or run-time system can also invalidate benchmark results. As mentioned in [10], it is possible for an operating system to time-slice a benchmark and check for other processes to run even if the benchmark is the only process in the system. It is also possible for system daemon processes to become active and compete with the benchmark for CPU time. The run-time system can interfere with the benchmark as well if, for example, it begins a background task to perform garbage collection.

Many of these effects can be avoided if system daemons can be disabled. If some problems still occur, wild distortions appearing in the results can be discarded if they are very infrequent. Some effects cannot be avoided, but will probably involve only a small cost. When running the Michigan

benchmarks under Unix, it was estimated that operating system interference consumed no more than 5% of the run time [10]. The Michigan approach was to repeat the tests several times and assume that the smallest running times for both the control and test loops represented the cases where system interference was minimal. The PIWG approach involves an average of running times with extreme values on both ends being discarded. This will tend to average the system costs into the results. Other approaches to measuring this effect or factoring it out are possible.