Technical Report
CMU/SEI-87-TR-028
ESD-TR-87-191

A Survey of Real-Time Performance Benchmarks for the Ada Programming Language

Patrick Donohoe

December 1987

Technical Report

CMU/SEI-87-TR-28
ESD-TR-87-191
December 1987

A Survey of Real-Time Performance Benchmarks for the Ada Programming Language

Patrick Donohoe

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler SIGNATURE ON FILE
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

A Survey of Real-Time Performance Benchmarks for the Ada Programming Language

Abstract. This survey provides a summary description of some of the major Ada benchmarks currently available and an evaluation of their applicability to the Ada Embedded Systems Testbed (AEST) Project at the Software Engineering Institute (SEI). The benchmarks discussed are: the University of Michigan benchmarks, the ACM Performance Issues Working Group (PIWG) benchmarks, and the prototype Ada Compiler Evaluation Capability (ACEC) of the Institute for Defense Analyses (IDA).

1. Introduction

The primary purpose of the Ada Embedded Systems Testbed (AEST) project at the Software Engineering Institute (SEI) is to develop a solid in-house support base of hardware, software, and personnel to permit the investigation of a wide variety of issues related to software development for real-time embedded systems. Two of the most crucial issues to be investigated are the extent and quality of the facilities provided by Ada run-time support environments. The SEI support base will make it possible to determine assessments of the readiness of the Ada language and Ada tools to develop embedded systems.

The AEST Project testbed is essentially a host-target environment. It is hoped that some, if not all, of the benchmarks may be used to determine Ada performance characteristics on the testbed target: initially a DEC MicroVAX II running the VAXELN real-time executive, and later a Motorola MC68020 microprocessor. A MIL-STD-1750A processor is also a possible future target.

This report covers the Ada benchmarks developed at the University of Michigan [3] and by the ACM SIGAda Performance Issues Working Group (PIWG).[1] The PIWG benchmarks include Ada versions of the Whetstone [4] and Dhrystone [7] synthetic benchmarks. The report also discusses briefly the prototype Ada Compiler Evaluation Capability (ACEC) [6].

2. The University of Michigan Ada Benchmarks

The University of Michigan benchmarks concentrate on techniques for measuring the performance of individual features of the Ada programming language. In addition, the performance of some run-time system features — scheduling and storage management — is measured. The development of the real-time performance measurement techniques and the interpretation of the benchmark results are based on the Ada notion of time. A paper by the Michigan team [3] begins by reviewing the Ada concept of time and the measurement techniques used in the benchmarks. The specific features measured are then discussed, followed by a summary and appraisal of the results obtained.

[1]The benchmarks came from the PIWG distribution tape known as TAPE_8_31_86. The name, address, and telephone number of the current chairperson of the PIWG can be found in Ada Letters, a bimonthly publication of SIGAda, the ACM Special Interest Group on Ada.

To isolate a specific feature for execution-time measurement, a typical benchmark program executes a control loop and a test loop, the loops differing only by the feature to be examined. Time readings are taken at the beginning and end of both loops. Theoretically, the difference in execution times of the two loops is the execution speed of the feature being measured. The two loops are executed many times in a larger loop, and the desired time measurement is obtained by averaging. However, care must be taken to prevent compiler optimizations from removing portions of the benchmark code. Such optimizations include removing constants or constant expressions from loops, eliminating subprogram calls, or removing the feature being measured. The techniques used by the Michigan team to thwart code optimizers are discussed in [3]. A good example of language feature measurement techniques and compiler optimization blocking is given in [2].

Obtaining accuracy in measurements is another important complication discussed in the Michigan paper. To determine the number of iterations needed to obtain a measurement within a given tolerance, knowledge of both the resolution of the CLOCK function and the variability of the time needed to take a time measurement is needed. In the clock calibration program, a call to the CLOCK function is placed in a loop that is executed a large number of times, and each time measurement obtained is placed in an array. A second differencing scheme applied to these measurements will yield the time resolution [3].

The Michigan paper also explains how problems such as isolating features to be measured, ensuring sufficient accuracy of measurements, avoiding operating system distortions, and obtaining repeatable results were handled. The rationale for each benchmark test is also explained in that paper.

The Ada language features measured by the benchmarks are:

- subprogram calls
- dynamic storage allocation
- exception handling
- task elaboration, activation, and termination
- task rendezvous
- CLOCK function resolution and overhead
- time math

The Ada runtime features measured by the benchmarks are:

- scheduling considerations (**delay** statement)
- object deallocation and garbage collection
- interrupt response time (outline only - see below)

The Michigan paper presents results for the following compilers:

- Verdix Versions 4.06, 5.1, and 5.2 running with UNIX 4.2 BSD on a VAX 11/780
- DEC VAX Ada Version 1.1 running with MicroVMS 4.1 on a MicroVAX II

- DEC VAX Ada Version 1.3 running with VMS 4.4 on a VAX 11/780
- Alsys Version 1.0 running with Aegis Version 9.2 on an Apollo DN 660

The authors of the Michigan paper point out that these versions of the compilers were intended for time-shared use, not real-time applications; hence, the results should not be interpreted with real-time performance in mind. The next two sections describe the various tests used to measure the language and runtime features.

2.1. Ada Language Features Measured

2.1.1. Subprogram Calls

Modular programming leads to increased use of subprograms and hence to increased call and return overhead. A way of reducing the overhead is to have the compiler generate an inline expansion of the subprogram; the tradeoff, however, is a larger object module. Ada has both subprogram calls and an INLINE pragma (but a compiler is not required to implement the latter). Measuring subprogram overhead versus the execution time of inline code may provide a basis for evaluating the tradeoff.

Several kinds of benchmarks are provided. They measure the overhead involved in entering and exiting a subprogram with no parameters, with various numbers of scalar parameters, and with various numbers of composite objects (arrays and records) as parameters. Tests are also provided to measure the overhead associated with passing constraint information to subprograms whose formal parameters are of an unconstrained composite type. All of the tests include passing parameters in all three modes: **in**, **out**, and **in out**.

All of the tests also measure the difference in overhead between calling subprograms in different packages and calling subprograms in the same package. For intra-package calls, there are also versions of the tests to measure the overhead of using the INLINE pragma if the pragma is supported. In these tests, the test loop calls an OVERHEAD procedure that has been specified by the INLINE pragma. The body of the OVERHEAD procedure is simply a call to a DO_NOTHING procedure. The control loop calls the DO_NOTHING procedure directly; the difference in execution speed between it and the test loop is a measure of the overhead involved in inline expansion of subprogram calls.

Finally, all the tests for inter- and intra-package calls are repeated with the subprograms appearing as part of a generic. These tests determine the overhead associated with executing generic instantiations of the code.

2.1.2. Dynamic Storage Allocation

Objects such as unconstrained arrays provide software flexibility and ease of support as an application and its storage needs change. However, dynamically allocating storage may be costly for critical real-time systems. To determine the feasibility of dynamic storage allocation in a real-time embedded application, the overhead involved must be measured.

There are three categories of allocation measured by available tests.

- *Fixed storage allocation*. The objects are declared locally in a subprogram or **declare** block; the storage required is known at compile time but is allocated at runtime.
- *Variable storage allocation*. Same as for fixed allocation, but the storage required (e.g., in the case of an array with variable bounds) is not known at compile time.
- *Explicit dynamic allocation*. Storage is allocated via the **new** allocator.

There are about 80 test programs to handle the first two cases, all of them variations on the same theme: allocating various numbers of integer, enumeration, string, and record objects, as well as arrays of various sizes. The naming convention used for these files is explained in the **READ_ME** file in the **documentation** subdirectory. For a typical example of how these tests are structured, and a sample of the file-naming scheme used, consider the test that allocates 10 INTEGER variables.

The main driver program for the test, the procedure DAD, is in file **dd_in10.a**. This file also contains the body of the package DAD_TIMERS, which contains the functions CONTROL_TIMER and DAD_TIMER. These functions are called from the driver program to execute the control and test loops. The heart of the test loop is executed by calling a procedure, DAD_DECLARE, which declares the 10 integers. The control loop in function CONTROL_TIMER declares the 10 integers directly, then calls a procedure DAD_NO_DECLARE. These procedures, necessary to defeat compiler optimizations, are contained in the body of package DAD_DECLARES in file **dd_in10B.a**. The specifications of the packages DAD_DECLARES and DAD_TIMERS are in file **dd_in10S.a**. The control loop in function CONTROL_TIMER also calls the procedure DAD_DO_SOMETHING to balance the code of the test loop. This procedure is in package DAD_TRIVIAL; the package specification is in file **dd_in10S.a**, and the body is in file **dd_in10B2.a**.

There are three main driver programs, **dan_1d.a**, **dan_2d.a**, and **dan_3d.a**, to handle the third category of storage allocation. They allocate one-, two-, and three-dimensional arrays whose sizes and types are specified in various supporting package specifications. The test loop in each program allocates an array of objects (integer, enumeration, string, array, record) using the **new** allocator.

2.1.3. Exception Handling

Error handling is a very important function of real-time embedded systems. For these systems to operate properly, efficient exception handling must be provided. To estimate the efficiency of Ada's exception handling, the time to respond to and propagate exceptions must be measured and examined.

One main program, **ex_main.a**, performs all the exception tests. It has a control loop that raises no exceptions, additional control in the form of an exception handler that never executes, and different versions of handlers for CONSTRAINT_ERROR, NUMERIC_ERROR, TASKING_ERROR, and user-defined exceptions. The times measured are as follows:

- the elapsed time between the raising of an exception and the start of execution of the exception handler in the same subprogram;
- the elapsed time between the raising of an exception in a subprogram and its subsequent raising in a calling subprogram (exception propagation time).

In these tests, exception handlers are written both as part of the main program and as part of subroutines called by the main program. The actual exceptions are generated both implicitly (e.g., add INTEGER'LAST to itself, causing a NUMERIC_ERROR) and explicitly (**raise** NUMERIC_ERROR;).

2.1.4. Task Elaboration, Activation, and Termination

Tasking is at the heart of the Ada programming language, but, as noted in [3], "... task elaboration, activation, and termination are almost always suspect operations in real-time programming, and programmers often allocate tasks statically to reduce runtime execution time." Thus, an indication of the time to perform these operations is of special interest.

The tasking tests measure the total time taken to elaborate a task's specification, activate the task, and terminate the task. According to the Michigan paper, the coarse resolution of the CLOCK function prevented the measurement of the individual times. The tests cover the two cases of task activation:

- *Entering the non-declarative part of a parent block*. There are two versions of this test. The first, **t_type_obj.a**, has the task type declared in the declarative region of the program, and an object of that task type declared and activated within the test loop of the program. The second, **t_dec_obj.a**, has the task object declared and activated directly within the test loop (inside a declare block).
- *Using the* **new** *allocator*. This test, **t_type_new.a**, declares a task type and a pointer to it in the declarative region of the program. A task object is then declared and allocated via the **new** allocator inside a declare block within the program's test loop.

In each case, the execution time of a control loop subtracted from the the time of the test loop yields the composite time for task elaboration, activation, and termination.

2.1.5. Task Rendezvous

One main program, **r_rend.a**, measures the time taken to complete a rendezvous between a procedure (the calling task) and a task. No parameters are passed with the entry call, and the acceptor task executes a simple **accept** statement. This test gives a lower bound on rendezvous time. The timing measurement includes the "cost" of at least two context switches. It is implied in [3] that other versions of the test exist for a rendezvous involving various numbers, types, and modes of passed parameters. The code for these versions of the test was not present on the tape received at the SEI; however, it would be simple to derive from the given main program. The generalized output routine provided for the tasking test can be used to print the results from any version of the test.

2.1.6. CLOCK Function Overhead

The CLOCK function provided by the CALENDAR package can be used extensively in applications where timed loops are required. The overhead associated with calling CLOCK may be a significant contribution to the total elapsed time of such loops. The benchmark program **co_main.a** measures the time required to perform a call to the CLOCK function.

2.1.7. TIME and DURATION Evaluations

For real-time systems coded in Ada, the need frequently arises to compute dynamically the sum or difference of a TIME value returned by the CLOCK function and a DURATION value; this computation might be used as the value in a **delay** statement. The actual delay experienced by the program may be longer than anticipated, depending upon the overhead involved in calling the CALENDAR package functions CLOCK, "+", and "-".

Fifteen versions of the same basic test program, **tm1.a** through **tm15.a**, provide measurements of the overhead involved in using the "+" and "-" functions of the CALENDAR package. (The test to measure CLOCK overhead has been referred to in the preceding subsection.) A simple assignment statement involving the sum or difference of objects of type TIME and/or DURATION is the essential feature of each test loop. The two versions of "+" contained in CALENDAR (to provide commutativity with respect to the operand types TIME and DURATION) are tested; although they are essentially the same, measurement discrepancies will arise if the CALENDAR package implements one of the functions as a call to the other.

2.2. Ada Runtime Features Measured

2.2.1. Delay and Scheduling Tests

Determining when a task becomes eligible for execution after the expiration of a **delay** statement is an important real-time programming issue. The Ada language reference manual states that order of scheduling for execution among tasks of equal or unstated priority is undefined, and that fair scheduling is presumed. An implementation may elect to check for the delay expiration periodically at synchronization points in a program or in a variety of other ways. Thus, a test is needed that will determine an implementation's scheduling discipline related to delay expiration.

A single test program, **dt_test.a**, measures the time taken to execute a **delay** statement for a range of delay values beginning with the value DURATION'SMALL. The Michigan paper [3] gives a comprehensive treatment of the scheduling considerations that motivated the benchmark (fixed-interval delay scheduling, time-slicing, pre-emptive), a discussion of the measurement techniques used by the test program, and a discussion of the interpretation of the test results.

2.2.2. Object Deallocation and Garbage Collection

The *Reference Manaual for the Ada Programming Language* [1] does not require the immediate return of deallocated storage to the storage pool; thus the implementation of garbage collection can have important consequences for embedded systems. Embedded systems are often required to run for lengthy periods of time; while the total amount of memory required at any one time may not be great, the system will run out of memory if deallocation does not occur. Even if deallocation does take place, the time taken for the garbage collector to recognize and respond to the need to release memory may be excessive for the needs of a particular real-time embedded system. Thus it is important to know the memory management characteristics of the runtime system being used.

There are four memory management test programs. The basic idea behind all four is to use the **new** allocator in a loop with various checks to determine whether or not garbage collection is possible.

1. **mm_memsize.a** allocates up to 10,000,000 integers in blocks of 1000 in an attempt to bump into the memory limit (i.e., raise STORAGE_ERROR). It prints the number of allocations actually performed, or an appropriate message if it does not hit the memory limit.
2. **mm_exp_deallocate.a** allocates memory exactly like the first program; however, the blocks of integers are explicitly deallocated using UNCHECKED_DEALLOCATION. This test will raise STORAGE_ERROR at the same point as in the first test (assuming it reaches the limit of memory) if UNCHECKED_DEALLOCATION is not implemented.
3. **mm_imp_deallocate.a** determines if the allocated blocks of integers are implicitly deallocated if no access variables are set to point to them. If no such automatic garbage collection is performed, STORAGE_ERROR will be raised as in the two previous tests.
4. **mm_first_diff.a** is an application of the second difference algorithm discussed in the Michigan paper [3]. It prints the first differences of the allocation times for 1000-integer blocks (arrays). This program is designed to test runtime effects in virtual memory systems; in such systems the amount of allocated memory can reach the point where paging takes place. By forcing memory to remain allocated and looping a sufficiently large number of times until STORAGE_ERROR is raised, the program can collect figures for memory allocation and paging times.

2.2.3. Interrupt Response Time

No programs were provided to measure interrupt response time; the Michigan paper explains why. Basically, the reasons are (a) the need (in general) for interrupt-generating hardware external to the CPU; and (b) the fact that times to be measured occur at very different points in a test program, so the use of iteration to improve measurement accuracy cannot be expected to work. A framework for the solution to the problem is provided in the Michigan paper.

3. The Performance Issues Working Group (PIWG) Ada Benchmarks

The PIWG benchmarks comprise more than 130 different tests that were either collected or developed by PIWG under the auspices of the ACM Special Interest Group on Ada (SIGAda). They can be grouped into three broad categories:

1. composite benchmarks
2. individual timing tests
3. compilation tests

The tests in each category are described below. Many of the individual timing tests are similar to the University of Michigan tests; in fact, the Michigan benchmarks are being added to the PIWG collection [3]. The PIWG benchmarks also use the same basic control and test loop structure.

3.1. Composite Benchmarks

3.1.1. The Whetstone Benchmark

The Whetstone benchmark [4], [5] is a single-program benchmark that was designed to reflect the frequency of language constructs used in real programs. The benchmark was originally developed to test Algol and FORTRAN compilers; the PIWG Ada version appears to be a straight translation of one of these older versions because it does not have any of the programming constructs found in newer languages (e.g., case statements). The program contains groups of statements ("modules") to perform the following:

- evaluation of expressions involving simple identifiers
- evaluation of expressions involving array elements
- procedure call with array as passed parameter
- conditional branching
- integer arithmetic
- use of trigonometric functions ATAN, SIN, and COS
- procedure call with simple identifiers as passed parameters
- array references and procedure calls without parameters
- more integer arithmetic
- use of standard math functions SQRT, EXP, LOG

Two versions of the PIWG Whetstone benchmark are provided. The first uses Ada mathematical routines coded as part of the benchmark program. The second uses the math library provided by the compiler vendor (or the host system, if the compiler vendor implements an interface to it).

3.1.2. The Dhrystone Benchmark

The Dhrystone benchmark [7] is a single-program benchmark that reflects the frequency of source language constructs used in real programs. It was designed as a more modern form of the Whetstone benchmark, reflecting constructs from newer programming languages. The Ada version, however, does not include features unique to Ada such as tasking or exception handling. Dhrystone contains 100 Ada statements that are balanced in terms of distribution of statement types, data types, and locality (global, local, parameter, and constant). The distribution of statement types within Dhrystone is based on recent statistics (references given in Weicker's paper) about the actual use of programming language features. The distribution is as follows:

Assignment statements:	53%
Control statements:	32%
Procedure and function calls:	15%

Weicker's paper lists tables of the frequency distribution data on which Dhrystone is based; it also discusses Pascal and C versions of the program and lists the complete text of the Ada program. Summarizing the use of Dhrystone, the paper says:

> The intention of this paper has been to present a better founded benchmark program for architecture or compiler discussions. The program has been used internally for comparisons of different microprocessors, for comparisons of micros with minicom-

puters, and for evaluation of experimental designs. In our experience, the results achieved with Dhrystone as a yardstick reflect fairly accurately the effectiveness of a particular hardware/compiler combination for systems programming applications.

3.1.3. The Hennessy Benchmark

This benchmark — named after the person who collected the programs — is a collection of well-known programming problems coded in Ada. They are all relatively small, both in terms of storage and execution speed. They can be used for comparing the Ada language with other programming languages.

The benchmark consists of a main program that calls the following subprograms:

- Matrix Multiplication
- Puzzle
- Treesort
- Permutations
- Towers of Hanoi
- Eight Queens Problem
- Quicksort
- Bubble Sort
- Fast Fourier Transform
- Ackermann's Function

3.2. Individual Language Feature Benchmarks

3.2.1. Task Creation and Termination

Three tests measure task creation and termination time. In the first test, the task body is elaborated in a package and activated from within a procedure contained in the same package. In the second test, the task is elaborated and activated from within a procedure inside a package. In the third test, the task is elaborated and activated locally within the test loop of the program.

3.2.2. Dynamic Storage Allocation

Four programs measure the allocation times of a 1000-integer array. The first simply allocates the array; the second allocates the array and initializes it using the **others** initialization feature. The third and fourth tests duplicate the first two, with the array defined as a field in a **record**.

3.2.3. Exception Handling

One program raises a user-defined exception and measures the time taken to enter the (null-bodied) exception. The second test has the exception handler in a different procedure from the one containing the **raise** statement; it measures the exception propagation delay. The third and final test is a version of the second, with the exception raised in a procedure nested four levels deep.

3.2.4. Coding Style

This test measures the difference in execution speeds between a Boolean condition coded as

```
FLAG := A < B;
```

versus the same condition coded as

```
if A < B then
   FLAG := TRUE;
else
   FLAG := FALSE;
end if;
```

3.2.5. TEXT_IO Procedures

The GET_LINE, GET, PUT, and PUT_LINE procedures of the package TEXT_IO are measured as they retrieve data from a file. Two additional GET tests retrieve an INTEGER and FLOAT value from a local string. A final test measures the overhead involved in opening a file.

3.2.6. Loop Overhead

Three tests measure the overhead associated with the three looping structures of Ada: the **for** loop, the **while** loop, and the infinite loop with an **exit** statement.

3.2.7. Subprogram Calls

There are 11 benchmarks to measure procedure call overhead. The following kinds of features are measured:

- procedures with no parameters:
    1. simple procedure - compiler may replace with inline code
    2. simple procedure - coding makes inline replacement impossible
    3. simple procedure in package
    4. simple procedure in package - **pragma** INLINE used
- procedures (in packages) with parameters:
    1. one **in** parameter
    2. one **out** parameter
    3. one **in out** parameter
    4. ten **in** parameters
    5. twenty **in** parameters
    6. ten **in** parameters, composite type (**record**)
    7. twenty **in** parameters, composite type (**record**)

3.2.8. Task Rendezvous

Six tests measure the rendezvous times of tasks with various numbers of **entry** calls, with and without a **select** statement. No parameters are passed during the rendezvous. All called tasks are variations of the following basic task body:

```
task body T1 is
begin
   loop
      accept E1 do
         .
         <code to defeat optimization>
         .
      end E1;
   end loop;
end;
```

One of the tests activates 10 single-entry tasks and computes an average rendezvous time.

3.3. Compilation Tests

There are over one hundred files in this category, mostly containing package specifications and bodies used to measure compilation speeds. There are two main groups of files:

Compile-link-execute. There are 24 packages that define Ada types and operators for physical quantities. For example, the package PHYSICAL_UNITS_ELECTRICAL defines such types as CURRENT_MILLIAMPERE and RESISTANCE_OHM. There are packages to convert between systems of measurement (e.g., MKS system of units to the English system), packages to define PUT procedures for the various units, and a package of physical constants. There is a program included which uses **with** and **use** clauses to pick up the information in all these packages. It solves a few simple physics problems involving a ball dropped from a height.

In addition to the physical quantities test, there are two others: one that is an attempt to create a file with at least one of each kind of Ada statement; and one containing a generic sort routine that is called to sort arrays of five different types: integer, float, string, enumeration, and fixed.

Compile-only tests. This is a collection of 69 null-bodied procedures that declare varying numbers of objects of just about every Ada type. For example, there are procedures to declare 100, 200, and 1000 integers; 100, 200, and 500 package specs; 100 tasks, and so on.

4. The Prototype Ada Compiler Evaluation Capability (ACEC)

The purpose of the Prototype ACEC is to provide users with an organized suite of compiler performance tests and support software for executing the tests and collecting performance statistics. The ACEC test suite was constructed by the Institute for Defense Analyses (IDA) for the Evaluation and Validation (E & V) Team of the Ada Joint Program Office (AJPO). The IDA has prepared a *User's Manual for the Prototype ACEC* [6]; most of the information listed here comes from that manual.

The 286 tests in the ACEC suite are organized into two major categories, based on the kind of information the test provides to the user. The categories are as follows:

- *Normative tests*. These provide information about language features that must be present in a compiler that claims to be a full implementation of ANSI/MIL-STD 1815A. There are two kinds of tests in this category:
    1. *Performance tests* collect speed and space attributes for various Ada language features;
    2. *Capacity tests* indicate limitations imposed by the compiler and runtime system on application developers (e.g., levels of recursion, size of stack).
- *Optional tests*. These may be selected by a user to represent an applications profile consisting of most frequently used language features. There are two subcategories:
    1. *Features tests* provide measurement of optional language features (those that are not a required part of an Ada compiler). They also provide measurements of the effects of certain compiling options.
    2. *Special algorithm tests* are combinations of language constructs that are characteristic of synthetic benchmark programs. They include such widely known benchmarks as Whetstone and the Sieve of Eratosthenes.

The *User's Manual for the Prototype ACEC* provides a more detailed description of the tests, how they perform data collection and evaluation; the overall software architecture (the database package, the report writer, the instrumentation package, and the actual benchmark tests); and how to run the benchmarks. A complete listing of all test file names and a one-line description of each test are also given.

Whether or not the Prototype ACEC tests will be useful to the AEST Project remains to be seen. They have already been used in the SEI Evaluation of Ada Environments Project; Chapter 8 of the project paper [8] says:

> The purpose of the experiment was to provide a method of evaluating Ada compilers, as a supplement to the main experiments of the project. The decision to make use of an existing test suite was driven by the desire to expend minimum effort in this supplementary activity. In retrospect, it has taken more effort than expected to employ the ACEC only to produce quite modest results.

As this survey report neared completion, it was learned that a contract to produce a full set of ACEC tests has been awarded to Boeing Military Airplane Company by the Evaluation and Validation team of the Ada Joint-Program Office. The SEI will participate in the evaluation of these tests.

5. Summary and Conclusions

Of the benchmarks discussed in this report, the two that appear to be most relevant to the AEST Project are the University of Michigan benchmarks and the PIWG benchmarks. The Michigan benchmarks, in particular, contain many tests relevant to embedded real-time systems, and the paper by the Michigan team [3] gives a good description of each test, an explanation of the techniques used, guidelines for the interpretation of results, tables of test results for each compiler-machine combination, and a summary discussion of each test result. The relevance of benchmarking to real-time embedded systems is stated in the concluding paragraph of the Michigan paper:

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

The PIWG benchmarks are an evolving suite and will eventually incorporate the Michigan tests. Reports on the status of the benchmarks will be presented at SIGAda meetings and PIWG workshops. A newsletter giving benchmark results for various compilers and targets will be distributed periodically.

References

[1] *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983*
Ada Joint Program Office, 1430 Broadway, New York, NY 10018, 1983.

[2] Bassman, M.J., Fisher, G.A. Jr., and Gargaro, A.
An Approach for Evaluating the Performance Efficiency of Ada Compilers.
In *Ada in Use, Proceedings of the International Ada Conference (Paris, France, May 14-16)*. 1985.

[3] Clapp, Russell M., et al.
Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 29(8):760-778, August, 1986.

[4] Curnow, H. J., and Wichmann, B. A.
A Synthetic Benchmark.
The Computer Journal 19(1):43-49, February, 1976.

[5] Harbaugh, S., and Forakis, J.
Timing Studies Using a Synthetic Whetstone Benchmark.
Ada Letters 4(2):23-34, 1984.

[6] Hook, A. A., et al.
User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC), Version 1.
Technical Report P-1879, Institute for Defense Analyses, October, 1985.

[7] Weicker, Reinhold P.
Dhrystone: A Synthetic Systems Programming Benchmark.
Communications of the ACM 27(10):1013-1030, October, 1984.

[8] Weiderman, N. H., and Habermann, A. N.
Evaluation of Ada Environments.
Technical Report CMU/SEI-87-TR-1, Software Engineering Institute, January, 1987.

Table of Contents