Technical Report

CMU/SEI-87-TR-027
ESD-TR-87-190

Ada Performance Benchmarks on the MicroVAX II: Summary and Results

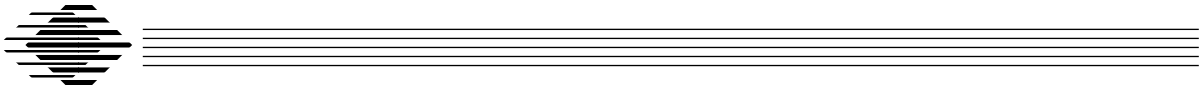
Version 1.0

Patrick Donohoe
December 1987

Technical Report
CMU/SEI-87-TR-27
ESD-TR-87-190
December 1987

Ada Performance Benchmarks on the MicroVAX II: Summary and Results

Version 1.0



Patrick Donohoe

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

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.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF SEI Joint Program Office

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

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA

15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Ada Performance Benchmarks on the MicroVAX II: Summary and Results Version 1.0

Abstract: This report documents the results obtained from running the University of Michigan and the ACM SIGAda Performance Issues Working Group (PIWG) Ada performance benchmarks on a DEC VAXELN MicroVAX II using the DEC VAXELN Ada compiler. A brief description of the benchmarks and the test environment is followed by a discussion of some problems encountered and lessons learned. The output of each benchmark program is also included.

1. Summary

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 runtime support environments. The SEI support base will make assessments possible of the readiness of the Ada language and Ada tools to develop embedded systems.

The benchmarking/instrumentation subgroup was formed to:

- Collect and run available Ada benchmark programs from a variety of sources on a variety of targets.
- Identify gaps in the coverage and fill them with new test programs.
- Review the measurement techniques used and provide new ones if necessary.
- Verify software timings by inspection and with specialized test instruments.

This report documents the results obtained from running Ada performance benchmarks on a DEC VAXELN MicroVAX II using the DEC VAXELN Ada compiler. The benchmarks were the University of Michigan Ada benchmarks and the ACM SIGAda Performance Issues Working Group (PIWG) Ada benchmarks (excluding the compilation tests). A description of these suites and the reasons for choosing them are given in [9]. The benchmarks focus largely on the execution time of specific features of the Ada language; they do not, for example, measure the efficiency or the size of the generated object code. A brief description of the benchmarks and the test environment is followed by a discussion of some problems encountered and lessons learned. The results obtained from running the entire Michigan and PIWG benchmark suites are contained in the appendices to this report. Note that the

caveats discussed in the body of the report must be borne in mind when examining these results.

2. Discussion

2.1. 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. The development of the real-time performance measurement techniques and the interpretation of the benchmark results are based on the Ada notion of time. An article by the Michigan team [4] 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 of the results obtained and an appraisal of these results. A follow-up letter about the Michigan benchmarks appears in [3].

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

The PIWG benchmarks comprise many different Ada performance tests that were either collected or developed by PIWG under the auspices of the ACM Special Interest Group on Ada (SIGAda). In addition to language feature tests similar to the Michigan benchmarks, the PIWG suite contains composite synthetic benchmarks such as Whetstone [5], [10]; Dhrystone [11]; and a number of tests to measure speed of compilation. PIWG distributes tapes of the benchmarks to interested parties and collects and publishes the results in a newsletter. Workshops and meetings are held during the year to discuss new benchmarks and suggestions for improvements to existing benchmarks.[1]

2.3. Testbed Hardware and Software

The hardware used for benchmarking was a DEC MicroVAX II host, running MicroVMS 4.4, linked to a MicroVAX II target. The target had five megabytes of RAM, a dual floppy disk drive, and was linked to the host via DECnet. Programs on the target machine ran under control of the VAXELN kernel, an executive providing job and process scheduling on a prioritized pre-emptive basis [6], [7]. The hardware and software can be summarized as follows:

Host: DEC MicroVAX II, running MicroVMS 4.4

Compiler: DEC VAXELN Ada, release 1.1 (DEC VAX Ada 1.3), ACVC 1.7

Target: DEC MicroVAX II with VAXELN 2.3, 5Mb RAM

[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.

The complete VAXELN tool kit is a software product for the development of real-time systems for VAX processors. It provides most of the standard VAX/VMS development tools, such as the VAX Ada Compilation System (ACS), and includes a VAXELN Ada runtime library and a VAXELN remote debugger. The remote debugger can be used to download and activate programs on the target, whether or not they have been compiled with the debugger option. The VAXELN Ada compiler [8] is substantially identical to the VAX Ada compiler, with the exception of some pragmas (e.g., VAXELN Ada does not support the TIME_SLICE pragma) and VAXELN Ada's lack of relative and indexed file support. The host-based development tools are used to create an application program and build a VAXELN executable target system that can be booted on the target machine from a floppy disk or tape, or downloaded to the target via DECnet.

2.4. Running the Benchmarks

Both the Michigan and PIWG benchmark suites contained command files for compiling and running the tests under VAX/VMS. The Michigan benchmarks had a command file for each category of tests (e.g., one for rendezvous tests, one for exception handling tests), whereas the PIWG suite had a single command file that could be adapted to run any test. The Michigan command files were run through a "pre-processor" command file that produced an expanded command file capable of building and downloading a VAXELN executable system. The benchmark output, which normally would have appeared on the target machine's console, was re-routed to a file on the host. It was also possible to create a bootable floppy disk; as a test, several executable VAXELN images were created as both a bootable floppy disk and a file to be downloaded from the host. Virtually no variation in the results produced by either method was observed, so the downloadable file became the preferred method since it could be fully controlled from the host.

All benchmarks were compiled with VAXELN Ada's default optimizations turned on.[2] The benchmarks contained code to prevent the language feature of interest from being optimized away. Runtime checks were not suppressed, and, apart from the Michigan exception-handling problem noted below, the benchmarks' source code was not modified in any way. Benchmark results are listed in the appendices.

2.5. Problems Encountered and Lessons Learned

A number of minor problems were encountered during the running of the benchmarks; these are noted below in the appropriate results section. The one major problem that arose only appeared after most of the Michigan tests had been run: negative time values were produced for some of the tests (Dynamic Storage Allocation and Subprogram Overhead tests). An investigation revealed that the VAXELN paging mechanism lengthened the ex-

[2]The compiler performs a number of standard optimizations, including: elimination of common subexpressions; removal of invariant computations from loops; in-line code expansion; global assignment of variables to registers; peephole optimization of instruction sequences; and elimination of dead code. If these optimizations are not desired, the user must explicitly disable them by invoking an option with the compile command.

ecution times of loops that spanned a page boundary. (Physical memory on the VAXELN target is divided into 512-byte pages; however, no swapping to disk took place since disk support was not included. The benchmarks were entirely resident in memory.) Thus the control loop of some benchmarks would actually take longer to run than the test loop, and the execution time of the language feature being measured (expressed as the difference of the test and control times) would sometimes be negative. A more detailed discussion of the so-called "dual loop problem" can be found in [1]. A complete report on the problems encountered during the AEST benchmarking effort, and a discussion of other possible benchmarking pitfalls, is contained in [2].

Another interesting issue is the accuracy of times reported by the PIWG benchmarks. One of the PIWG benchmark support packages, A000032.ADA, contains the body of the ITERATION package. This package is called by a benchmark program to calculate, among other things, the minimum duration for the test loop of a benchmark run. The minimum duration is computed to be the larger of 1 second, 100 times **System.Tick**, and 100 times **Standard.Duration'Small**. The idea appears to be (a) to run the benchmark for enough iterations to overcome the problem of the relatively coarse resolution of the **Calendar.Clock** function, and (b) to provide a relative accuracy of one percent or better. The times reported by the benchmark programs are printed with an accuracy of one tenth of a microsecond; however, merely running the test for a specific minimum duration does not guarantee this degree of accuracy. If the clock resolution is 10 milliseconds, for example, and the desired accuracy is to within 1 microsecond, then the test should be run for 10,000 iterations. For Ada language features that execute in tens of microseconds, running for a specific duration may ensure enough iterations for accuracy to within one microsecond; this is not so for language features that take longer.

In general, the accuracy of the PIWG and Michigan benchmarks is to within one tick of **Calendar.Clock** divided by the number of iterations of the benchmark (see the Basic Measurement Accuracy section of the University of Michigan report). The University of Michigan benchmarks typically run for 10,000 iterations, and so are accurate to within 1 microsecond for VAXELN Ada (10 millisecond **Calendar.Clock** resolution). The task creation tests and some of the dynamic storage allocation tests run for fewer iterations, probably because of the amount of storage they use up; the reduced accuracy is noted in the appropriate sections. Also, the source of the exception-handling tests had to be modified to reduce the number of iterations so that the test would actually run. For the PIWG tests, a table of iteration counts and resultant accuracy is provided in the PIWG results appendix.

Comparison of the results from the most closely equivalent PIWG and Michigan benchmarks has been hindered by the accuracy problem and the dual loop problem. Even when the correction factors are applied to take care of the former, the precise effects of the dual loop problem on each benchmark program are not known. It is clear that more work needs to be done to resolve such problems.

The VAXELN benchmarking effort was essentially a learning experience. The major lessons learned were:

- It is very important to check the underlying assumptions incorporated in the benchmark design before attempting to use the benchmark. A simple example of such a check is a "calibration" routine to check whether or not a dual loop test with textually identical loops will zero out.
- Even when few or no problems are encountered during the running of the benchmarks, the results should be checked for reasonableness, especially if the times reported are different from heuristically calculated figures.
- Inspection of generated assembly code (however distasteful this might be to an Ada aficionado) can turn up clues to puzzling results. Once problems start occurring, knowledge of the machine's instruction set architecture and underlying hardware can prove very useful.

The major result of the VAXELN MicroVAX benchmarking effort, therefore, is not a list of numbers to be taken at face value; rather, it is an appreciation of the problems and pitfalls facing the would-be benchmarker. Analysis of the results from the VAXELN and other cross-compilers and target systems, as well as analysis of the benchmarks themselves, will be one of the main items of business in the AEST Project's second year.

References

[1] Altman, N. A., and Weiderman, N. H.
Timing Variation in Dual Loop Benchmarks.
Technical Report SEI-87-TR-21, Software Engineering Institute, September, 1987.

[2] Altman, N. A.
Factors Causing Unexpected Variations in Ada Benchmarks.
Technical Report SEI-87-TR-22, Software Engineering Institute, September, 1987.

[3] Broido, Michael D.
Response to Clapp et al: Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 30(2):169-171, February, 1987.

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

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

[6] *VAXELN User's Guide.*
Digital Equipment Corp., 1985.

[7] *VAXELN Release Notes.*
Digital Equipment Corp., 1986.

[8] *VAXELN Ada User's Manual.*
Digital Equipment Corp., 1986.

[9] Donohoe, P.
A Survey of Real-Time Performance Benchmarks for the Ada Programming Language.
Technical Report SEI-87-TR-28, Software Engineering Institute, December, 1987.

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

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

Appendix A: Results: University of Michigan Benchmarks

In the results presented below, certain lines of output have been omitted for the sake of brevity. Many of the Michigan tests print out lines of "raw data," and the command files sometimes run a particular test many times; these are the lines that have been omitted. Also, some of the headings have been split over two lines to make them fit this document.

A.a. Clock Calibration and Overhead

One of the Michigan "tests" merely prints the values of **System.Tick** and **Standard.Duration'Small**. For VAXELN Ada these are:

```
System Tick=         0.009948730468750  seconds
Duration Small=      0.000061035156250  seconds
```

Thus **System.Tick** is approximately 10 milliseconds, and **Duration'Small** is approximately 61 microseconds. The clock calibration test determines the resolution of the **Calendar.Clock** function. As can be seen from the data below, the resolution is 10 milliseconds, the value of **System.Tick**.

```
Output of second differencing is as follows:
Number zeros previous:                 94
Time difference (in seconds):        0.009948730468750
Number zeros previous:                  0
Time difference (in seconds):       -0.009948730468750
Number zeros previous:                112
Time difference (in seconds):        0.009948730468750
            .                              .
            .                              .
            .                              .

Number zeros previous:                112
Time difference (in seconds):        0.009948730468750
Number zeros previous:                  0
Time difference (in seconds):       -0.009948730468750
Number of iterations = 10000
```

It should be noted that the negative times above are a legitimate result of the test and have nothing to do with the dual loop problem discussed earlier.

The test to measure the overhead associated with calling **Calendar.Clock** produced consistently repeatable results, so only one line of output is shown:

```
Clock function calling overhead :   84.00  microseconds
```

A.b. Task Rendezvous

For this test, a procedure calls the single entry point of a task; no parameters are passed, and the called task executes a simple **accept** statement. According to the Michigan report, it is assumed that such a rendezvous will involve at least two context switches.

```
Rendezvous time : No parameters passed
Number of iterations = 10000

Task rendezvous time :     1585.0 microseconds
```

A.c. Task Creation

These tests measure the composite time taken to elaborate a task's specification, activate the task, and terminate the task. The coarse resolution of the clocks available at the time the tests were developed did not allow for measurement of the individual components of the test. Also, because these tests are run for 100 iterations, the reported times are accurate to 100 microseconds, or 0.1 milliseconds.

To obtain the third test result below, the VAXELN pool size (which determines the number of VAXELN objects that can be in simultaneous use) had to be increased from the default of 384 blocks to 1024 blocks (a block is 512 bytes).

```
Task elaborate, activate, and terminate time:
Declared object, no type
Number of iterations = 100

Task elaborate, activate, terminate time:     9.7 milliseconds
```

```
Task elaborate, activate, and terminate time:
Declared object, task type
Number of Iterations = 100

Task elaborate, activate, terminate time:     9.5 milliseconds
```

```
Task elaborate, activate, and terminate time:
NEW object, task type
Number of iterations = 100

Task elaborate, activate, terminate time:     8.9 milliseconds
```

A.d. Exception Handling

The exception-handling benchmark kept crashing with a STORAGE_ERROR exception despite many attempts to tailor the storage parameters of the VAXELN system build process. Eventually it was made to run by reducing the number of iterations of the test from 1000 to 100. This was the only case where benchmark code had to be modified. A possible reason for the problem (see the Memory Management section) is the lack of storage reclamation (garbage collection) procedures; space used during exception-handling probably remains allocated after the exception-raising procedure exits. The reduced number of iterations means that the times shown below are accurate only to within 100 microseconds.

```
Number of iterations = 100


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

      Exception raised and handled in a block

         0.0 uSEC.    User defined, not raised
       799.6 uSEC.    User defined
       999.8 uSEC.    Constraint error, implicitly raised
       999.8 uSEC.    Constraint error, explicitly raised
       499.9 uSEC.    Numeric error, implicitly raised
       999.8 uSEC.    Numeric error, explicitly raised
       999.8 uSEC.    Tasking error, explicitly raised

      Exception raised in a procedure and handled in the
      calling unit

         0.0 uSEC.    User defined, not raised
       900.3 uSEC.    User defined
      1000.4 uSEC.    Constraint error, implicitly raised
      1000.4 uSEC.    Constraint error, explicitly raised
       800.2 uSEC.    Numeric error, implicitly raised
      1000.4 uSEC.    Numeric error, explicitly raised
      1000.4 uSEC.    Tasking error, explicitly raised
```

A.e. Time and Duration Math

In the results below, the lines flagged with an asterisk are from tests that had to be run individually to get them to work. When included in a command file that ran all of the tests sequentially, these two tests would always cause VAXELN Ada to generate a runtime error message saying that the "computed year is not in the range of subtype YEAR_NUMBER."

```
Number of Iterations = 10000


          Time and Duration Math
          ======================

    uSEC.   Operation

    90.00   Time := Var_time + var_duration
    94.00   Time := Var_time + const_duration
    89.00   Time := Var_duration + var_time
    94.00   Time := Const_duration + var_time
*   93.00   Time := Var_time - var_duration
*   94.00   Time := Var_time - const_duration
   103.00   Duration := Var_time - var_time
     3.00   Duration := Var_duration + var_duration
     3.00   Duration := Var_duration + const_duration
     3.00   Duration := Const_duration + var_duration
     4.00   Duration := Const_duration + const_duration
     3.00   Duration := Var_duration - var_duration
     4.00   Duration := Var_duration - const_duration
     3.00   Duration := Const_duration - var_duration
     3.00   Duration := Const_duration - const_duration
```

A.f. Delay Statement Tests

For VAXELN Ada, **System.Tick** is 10 milliseconds and **Standard.Duration'Small** is 61 microseconds. In the results below, the desired delay times start at **Duration'Small** and increment by **Duration'Small**. The actual delay time of 0.01996 seconds is twice **System.Tick**; 0.02997 is three times **System.Tick**; and 0.03998 is four times **System.Tick**. Thus the smallest delay that can be achieved by a **delay** statement in the VAXELN implementation is approximately 20 milliseconds.

```
Number of iterations = 1

For case number           1
Desired delay time:     0.00006 seconds
Actual delay time:      0.01996 seconds

For case number           2
Desired delay time:     0.00012 seconds
Actual delay time:      0.01996 seconds
       .                      .
       .                      .
       .                      .

For case number         164
Desired delay time:     0.01001 seconds
Actual delay time:      0.01996 seconds

For case number         165
Desired delay time:     0.01007 seconds
Actual delay time:      0.02997 seconds
       .                      .
       .                      .
       .                      .

For case number         328
Desired delay time:     0.02002 seconds
Actual delay time:      0.02997 seconds

For case number         329
Desired delay time:     0.02008 seconds
Actual delay time:      0.03998 seconds
       .                      .
       .                      .
       .                      .

For case number         350
Desired delay time:     0.02136 seconds
Actual delay time:      0.03998 seconds
```

A.g. Dynamic Storage Allocation

There are three categories of allocation measured by these tests:

1. 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 run time.
2. 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.
3. Explicit Dynamic Allocation: Storage is allocated via the **new** allocator.

These tests were the first to exhibit symptoms of the "dual loop" problem (negative times) referred to earlier in this report.

Number of iterations = 10000

Dynamic Allocation in a Declarative Region

Time (microsec.)	# Declared	Type Declared	Size of Object
-5.0	1	Integer	
-1.0	10	Integer	
-16.0	100	Integer	
-3.0	1	String	1
-3.0	1	String	10
-3.0	1	String	100
-1.0	1	Enumeration	
-2.0	10	Enumeration	
-26.0	100	Enumeration	
-4.0	1	Integer array	1
-4.0	1	Integer array	10
-1.0	1	Integer array	100
-2.0	1	Integer array	1000
13.0	1	1-D Dynamically bounded array	1
22.0	1	1-D Dynamically bounded array	10
19.0	1	2-D Dynamically bounded array	1
25.0	1	2-D Dynamically bounded array	100
42.0	1	3-D Dynamically bounded array	1
41.0	1	3-D Dynamically bounded array	1000
-5.0	1	Record of integer	1
-4.0	1	Record of integer	10
-1.0	1	Record of integer	100

Because these tests only iterate 1000 times, the reported times are accurate to within 10 microseconds, rather than 1 microsecond.

Number of iterations = 1000

Dynamic Allocation with NEW allocator

Time (microsec.)	# Declared	Type Declared	Size of Object
280.0	1	Integer	1
280.0	1	Enumeration	1
280.0	1	Record of integer	1
290.0	1	Record of integer	10
280.0	1	Record of integer	100
280.0	1	Record of integer	20
290.0	1	Record of integer	5
290.0	1	Record of integer	50
290.0	1	Integer array	1
290.0	1	Integer array	10
290.0	1	Integer array	100
290.0	1	Integer array	1000
290.0	1	String	1
290.0	1	String	10
300.0	1	String	100
310.0	1	1-D Dynamically bounded array	1
310.0	1	1-D Dynamically bounded array	10
340.0	1	2-D Dynamically bounded array	1
340.0	1	2-D Dynamically bounded array	100
390.0	1	3-D Dynamically bounded array	1
390.0	1	3-D Dynamically bounded array	1000

A.h. Subprogram Overhead

Several kinds of subprogram overhead 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.[3] 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.

The subprogram overhead tests were the second major source of negative time values. The negative numbers for these tests were generally a lot smaller than those produced by the dynamic storage allocation tests.

Subprogram Overhead (non-generic)

Number of iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
0.8		0		
0.2	I	1	INTEGER	
0.0	O	1	INTEGER	
0.7	I_O	1	INTEGER	
-0.1	I	10	INTEGER	
0.1	O	10	INTEGER	
13.2	I_O	10	INTEGER	
134.6	I	100	INTEGER	
197.4	O	100	INTEGER	
303.6	I_O	100	INTEGER	

continued ...

[3] VAXELN Ada supports the INLINE pragma.

-0.2	I	1	ENUMERATION		
0.0	O	1	ENUMERATION		
0.6	I_O	1	ENUMERATION		
0.4	I	10	ENUMERATION		
-1.4	O	10	ENUMERATION		
2.0	I_O	10	ENUMERATION		
135.3	I	100	ENUMERATION		
188.8	O	100	ENUMERATION		
294.5	I_O	100	ENUMERATION		
1.7	I	1	ARRAY of INTEGER	1	
-1.8	O	1	ARRAY of INTEGER	1	
-0.1	I_O	1	ARRAY of INTEGER	1	
0.1	I	1	ARRAY of INTEGER	10	
0.0	O	1	ARRAY of INTEGER	10	
0.8	I_O	1	ARRAY of INTEGER	10	
-1.2	I	1	ARRAY of INTEGER	100	
0.0	O	1	ARRAY of INTEGER	100	
0.4	I_O	1	ARRAY of INTEGER	100	
0.2	I	1	RECORD of INTEGER	1	
0.1	O	1	RECORD of INTEGER	1	
0.2	I_O	1	RECORD of INTEGER	1	
-0.4	I	1	RECORD of INTEGER	100	
0.5	O	1	RECORD of INTEGER	100	
2.8	I_O	1	RECORD of INTEGER	100	
-0.2	I	1	UNCONSTRAINED ARRAY	1	
-0.2	O	1	UNCONSTRAINED ARRAY	1	
1.5	I_O	1	UNCONSTRAINED ARRAY	1	
-0.3	I	1	UNCONSTRAINED ARRAY	100	
-0.3	O	1	UNCONSTRAINED ARRAY	100	
0.1	I_O	1	UNCONSTRAINED ARRAY	100	
-1.2	I	1	UNCONSTRAINED RECORD	1	
0.2	O	1	UNCONSTRAINED RECORD	1	
0.1	I_O	1	UNCONSTRAINED RECORD	1	
0.1	I	1	UNCONSTRAINED RECORD	100	
-0.4	O	1	UNCONSTRAINED RECORD	100	
0.1	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
0.9		0		
0.3	I	1	INTEGER	
-0.1	O	1	INTEGER	
0.7	I_O	1	INTEGER	
0.1	I	10	INTEGER	
-0.2	O	10	INTEGER	
13.2	I_O	10	INTEGER	
134.5	I	100	INTEGER	
197.5	O	100	INTEGER	
303.9	I_O	100	INTEGER	
-0.2	I	1	ENUMERATION	
-0.1	O	1	ENUMERATION	
0.7	I_O	1	ENUMERATION	
0.2	I	10	ENUMERATION	
-1.5	O	10	ENUMERATION	
2.1	I_O	10	ENUMERATION	
135.2	I	100	ENUMERATION	
188.6	O	100	ENUMERATION	
294.2	I_O	100	ENUMERATION	
1.7	I	1	ARRAY of INTEGER	1
-1.9	O	1	ARRAY of INTEGER	1
-0.1	I_O	1	ARRAY of INTEGER	1
0.0	I	1	ARRAY of INTEGER	10
-0.4	O	1	ARRAY of INTEGER	10
0.9	I_O	1	ARRAY of INTEGER	10
-1.4	I	1	ARRAY of INTEGER	100
-0.2	O	1	ARRAY of INTEGER	100
0.4	I_O	1	ARRAY of INTEGER	100
0.0	I	1	RECORD of INTEGER	1
0.1	O	1	RECORD of INTEGER	1
0.1	I_O	1	RECORD of INTEGER	1
-0.6	I	1	RECORD of INTEGER	100
0.6	O	1	RECORD of INTEGER	100
2.9	I_O	1	RECORD of INTEGER	100

...continued

```
|    0.1   |   I     |      1 |UNCONSTRAINED ARRAY |       1  |
|   -0.2   |   O     |      1 |UNCONSTRAINED ARRAY |       1  |
|    1.5   |   I_O   |      1 |UNCONSTRAINED ARRAY |       1  |
|   -0.5   |   I     |      1 |UNCONSTRAINED ARRAY |     100  |
|   -0.4   |   O     |      1 |UNCONSTRAINED ARRAY |     100  |
|    0.0   |   I_O   |      1 |UNCONSTRAINED ARRAY |     100  |
|   -1.4   |   I     |      1 |UNCONSTRAINED RECORD|       1  |
|    0.3   |   O     |      1 |UNCONSTRAINED RECORD|       1  |
|    0.0   |   I_O   |      1 |UNCONSTRAINED RECORD|       1  |
|   -0.2   |   I     |      1 |UNCONSTRAINED RECORD|     100  |
|   -0.6   |   O     |      1 |UNCONSTRAINED RECORD|     100  |
|   -0.1   |   I_O   |      1 |UNCONSTRAINED RECORD|     100  |
```

Subprogram Overhead (non-generic, cross package)

Number of iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
39.4		0		
42.8	I	1	INTEGER	
45.8	O	1	INTEGER	
41.1	I_O	1	INTEGER	
43.4	I	10	INTEGER	
73.2	O	10	INTEGER	
108.7	I_O	10	INTEGER	
285.1	I	100	INTEGER	
472.0	O	100	INTEGER	
866.4	I_O	100	INTEGER	
42.2	I	1	ENUMERATION	
45.7	O	1	ENUMERATION	
41.1	I_O	1	ENUMERATION	
43.9	I	10	ENUMERATION	
72.0	O	10	ENUMERATION	
107.7	I_O	10	ENUMERATION	
271.4	I	100	ENUMERATION	
463.1	O	100	ENUMERATION	
847.9	I_O	100	ENUMERATION	
42.8	I	1	ARRAY of INTEGER	1
42.7	O	1	ARRAY of INTEGER	1
39.1	I_O	1	ARRAY of INTEGER	1
44.1	I	1	ARRAY of INTEGER	10
42.4	O	1	ARRAY of INTEGER	10
37.9	I_O	1	ARRAY of INTEGER	10
55.7	I	1	ARRAY of INTEGER	100
56.7	O	1	ARRAY of INTEGER	100
51.2	I_O	1	ARRAY of INTEGER	100
43.6	I	1	RECORD of INTEGER	1
42.9	O	1	RECORD of INTEGER	1
38.8	I_O	1	RECORD of INTEGER	1
56.2	I	1	RECORD of INTEGER	100
55.6	O	1	RECORD of INTEGER	100
52.1	I_O	1	RECORD of INTEGER	100

...continued

```
|   54.3   |   I     |      1 |UNCONSTRAINED ARRAY |       1  |
|   58.9   |   O     |      1 |UNCONSTRAINED ARRAY |       1  |
|   49.8   |   I_O   |      1 |UNCONSTRAINED ARRAY |       1  |
|   67.5   |   I     |      1 |UNCONSTRAINED ARRAY |     100  |
|   71.8   |   O     |      1 |UNCONSTRAINED ARRAY |     100  |
|   62.5   |   I_O   |      1 |UNCONSTRAINED ARRAY |     100  |
|   42.6   |   I     |      1 |UNCONSTRAINED RECORD|       1  |
|   43.9   |   O     |      1 |UNCONSTRAINED RECORD|       1  |
|   38.8   |   I_O   |      1 |UNCONSTRAINED RECORD|       1  |
|   55.3   |   I     |      1 |UNCONSTRAINED RECORD|     100  |
|   56.1   |   O     |      1 |UNCONSTRAINED RECORD|     100  |
|   52.1   |   I_O   |      1 |UNCONSTRAINED RECORD|     100  |
```

Subprogram Overhead (generic)

Number of iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
-0.3		0		
-5.3	I	1	INTEGER	
0.6	O	1	INTEGER	
0.5	I_O	1	INTEGER	
0.0	I	10	INTEGER	
0.1	O	10	INTEGER	
17.8	I_O	10	INTEGER	
112.9	I	100	INTEGER	
199.1	O	100	INTEGER	
304.4	I_O	100	INTEGER	
-4.9	I	1	ENUMERATION	
1.8	O	1	ENUMERATION	
-0.4	I_O	1	ENUMERATION	
-0.4	I	10	ENUMERATION	
-0.1	O	10	ENUMERATION	
10.1	I_O	10	ENUMERATION	
103.8	I	100	ENUMERATION	
191.7	O	100	ENUMERATION	
295.2	I_O	100	ENUMERATION	
-4.5	I	1	ARRAY of INTEGER	1
0.0	O	1	ARRAY of INTEGER	1
0.1	I_O	1	ARRAY of INTEGER	1
-2.9	I	1	ARRAY of INTEGER	10
0.1	O	1	ARRAY of INTEGER	10
0.8	I_O	1	ARRAY of INTEGER	10
-4.1	I	1	ARRAY of INTEGER	100
0.1	O	1	ARRAY of INTEGER	100
0.0	I_O	1	ARRAY of INTEGER	100
-4.4	I	1	RECORD of INTEGER	1
0.0	O	1	RECORD of INTEGER	1
0.0	I_O	1	RECORD of INTEGER	1
-3.9	I	1	RECORD of INTEGER	100
0.0	O	1	RECORD of INTEGER	100
0.0	I_O	1	RECORD of INTEGER	100

Subprogram Overhead (generic, cross package)

Number of iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
14.3		0		
15.1	I	1	INTEGER	
19.8	O	1	INTEGER	
24.6	I_O	1	INTEGER	
23.7	I	10	INTEGER	
51.6	O	10	INTEGER	
89.5	I_O	10	INTEGER	
277.4	I	100	INTEGER	
442.2	O	100	INTEGER	
831.5	I_O	100	INTEGER	
14.4	I	1	ENUMERATION	
19.1	O	1	ENUMERATION	
24.7	I_O	1	ENUMERATION	
25.8	I	10	ENUMERATION	
52.2	O	10	ENUMERATION	
89.3	I_O	10	ENUMERATION	
281.6	I	100	ENUMERATION	
422.5	O	100	ENUMERATION	
814.2	I_O	100	ENUMERATION	
14.4	I	1	ARRAY of INTEGER	1
15.5	O	1	ARRAY of INTEGER	1
19.4	I_O	1	ARRAY of INTEGER	1
20.7	I	1	ARRAY of INTEGER	10
25.3	O	1	ARRAY of INTEGER	10
22.4	I_O	1	ARRAY of INTEGER	10
21.9	I	1	ARRAY of INTEGER	100
25.0	O	1	ARRAY of INTEGER	100
23.8	I_O	1	ARRAY of INTEGER	100
16.1	I	1	RECORD of INTEGER	1
19.7	O	1	RECORD of INTEGER	1
19.6	I_O	1	RECORD of INTEGER	1
21.9	I	1	RECORD of INTEGER	100
24.1	O	1	RECORD of INTEGER	100
23.8	I_O	1	RECORD of INTEGER	100

A.i. Memory Management

There are no timing results produced by these tests; they are used to determine whether or not garbage collection takes place. They attempt to allocate up to ten million integers by successively allocating 1000-integer arrays using the **new** allocator. Only the last test explicitly attempted to free any allocated storage (using UNCHECKED_DEALLOCATION). The tests were designed either to report how much storage they allocated before the expected STORAGE_ERROR exception occurred, or a message saying they had succeeded. Running the tests confirmed that garbage collection did not occur; reclamation of storage is only done when explicitly requested. This may be the reason why the exception-handling tests would not run until the number of iterations was reduced (see the Exception Handling section).

An additional test included with the memory management tests uses a first differencing scheme to determine the scheduling discipline of the target operating system. This test was not run because it was already known that VAXELN is a pre-emptive priority-based system.

Appendix B: Results: PIWG Benchmarks

All of the PIWG tests, with the exception of the Hennessy benchmark (see below), ran without problems and without the need to tailor the VAXELN system-build process. The G tests (Text_IO tests) and the Z tests (compilation tests) were not run. None of the PIWG tests produced negative numbers.

The output of each PIWG benchmark program contains a terse description of the feature being measured. For any further details, the user will have to inspect the benchmark code. The reported "Wall Time" is based on calls to the **Calendar.Clock** function. The reported "CPU-Time" is based on calls to the PIWG function CPU_TIME_CLOCK. This function is intended to provide an interface to host-dependent CPU-time measurement functions on multi-user systems where calls to **Calendar.Clock** might return misleading results. For the VAXELN MicroVAX tests, the basic version of CPU_TIME_CLOCK, which simply calls **Calendar.Clock**, was used.

Because of the issue of the accuracy of PIWG results (see Problems Encountered and Lessons Learned section), the table below is provided. Note that the actual iterations of the benchmarks are 100 times greater than the reported iteration counts. The reported counts are only for the main loop enclosing the control and test loops; these latter loops alway iterate 100 times. The accuracy delta is computed by dividing the resolution of the **Calendar.Clock** function (10 milliseconds) by the actual number of iterations.

Reported Iteration Count	Actual Iterations	Accuracy Delta in Microseconds
1	100	100.0
2	200	50.0
4	400	25.0
8	800	12.5
16	1600	6.25
32	3200	3.125
64	6400	1.5625
128	12800	0.781250
256	25600	0.390625

B.a. Composite Benchmarks

B.0.0.1. The Dhrystone Benchmark

This is a version of the benchmark described in [11].

```
1.1710 is time in milliseconds for one Dhrystone
```

B.0.0.2. The Whetstone Benchmark

Two versions of the Whetstone benchmark [5] are provided. One uses the math library supplied by the vendor (**with** FLOAT_MATH_LIB for the VAXELN Ada compiler); the other has the math functions coded within the benchmark program so that the test can be run even when a math library is not supplied. "KWIPS" means Kilo Whetstones Per Second.

```
ADA Whetstone benchmark
A000092 using manufacturer's math routines

Average time per cycle :   808.32 milliseconds
 Average Whetstone rating :        1237 KWIPS


ADA Whetstone benchmark
A000093 using standard internal math routines

Average time per cycle :  1046.63 milliseconds
 Average Whetstone rating :         955 KWIPS
```

B.0.0.3. The Hennessy Benchmark

This is a collection of benchmarks that are relatively short in terms of program size and execution time. Named after the person who gathered the tests, it includes such well-known programming problems as the Eight Queens problem, the Tower of Hanoi, Quicksort, Bubble Sort, Fast Fourier Transform, and Ackermann's Function. The Hennessy benchmark, known as PIWG A000094, was the only PIWG benchmark that failed to execute; it crashed with a STORAGE_ERROR exception. Initial attempts to resolve the problem were unsuccessful. It is believed, however, that the solution lies in simply finding the right settings for the storage parameters of the VAXELN build process.

B.b. Task Creation

```
Test name:   C000001                        Class name:  Tasking
CPU time:     9400.0  microseconds
Wall time:    9400.0  microseconds         Iteration count:   2
Test description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop


Test name:   C000002                        Class name:  Tasking
CPU time:     9549.9  microseconds
Wall time:    9549.9  microseconds         Iteration count:   2
Test description:
 Task create and terminate time measurement
 with one task, no entries, when task is in a procedure
 task defined and used in procedure, no select statement, no loop


Test name:   C000003                        Class name:  Tasking
CPU time:     9599.9  microseconds
Wall time:    9599.9  microseconds         Iteration count:   2
Test description:
 Task create and terminate time measurement
 task is in declare block of main procedure
 one task, no entries, task is in the loop
```

B.c. Dynamic Storage Allocation

```
Test name:   D000001                        Class name:  Allocation
CPU time:       38.3  microseconds
Wall time:      38.3  microseconds         Iteration count:   128
Test description:
 Dynamic array allocation, use and deallocation time measurement
 dynamic array elaboration, 1000 integers in a procedure
 get space and free it in the procedure on each call


Test name:   D000002                        Class name:  Allocation
CPU time:     4225.0  microseconds
Wall time:    4225.0  microseconds         Iteration count:   4
Test description:
 Dynamic array elaboration and initialization time measurement
 allocation, initialization, use and deallocation
 1000 integers initialized by others=>1


Test name:   D000003                        Class name:  Allocation
CPU time:       23.4  microseconds
Wall time:      23.4  microseconds         Iteration count:   128
Test description:
 Dynamic record allocation and deallocation time measurement
 elaborating, allocating and deallocating
 record containing a dynamic array of 1000 integers


Test name:   D000004                        Class name:  Allocation
CPU time:     5350.3  microseconds
Wall time:    5350.3  microseconds         Iteration count:   2
Test description:
 Dynamic record allocation and deallocation time measurement
 elaborating, initializing by (DYNAMIC_SIZE,(others=>1))
 record containing a dynamic array of 1000 Integers
```

B.d. Exception Handling

There is no E000003 test in the PIWG 8/31/86 suite.

```
Test name:   E000001                          Class name:  Exception
CPU time:      825.0  microseconds
Wall time:     825.0  microseconds         Iteration count:   16
Test description:
 Time to raise and handle an exception
 exception defined locally and handled locally


Test name:   E000002                          Class name:  Exception
CPU time:     1093.8  microseconds
Wall time:    1093.8  microseconds         Iteration count:   16
Test description:
 Exception raise and handle timing measurement
 when exception is in a procedure in a package


Test name:   E000004                          Class name:  Procedure
CPU time:      881.2  microseconds
Wall time:     881.2  microseconds         Iteration count:   16
Test description:
 Exception raise and handle timing measurement
 when exception is in a package four deep
```

B.e. Coding Style

```
Test name:   F000001                        Class name:  Style
CPU time:        3.9  microseconds
Wall time:       4.3  microseconds         Iteration count:   256
Test description:
 Time to set a boolean flag using a logical equation
 a local and a global integer are compared
 compare this test with F000002


Test name:   F000002                        Class name:  Style
CPU time:        2.7  microseconds
Wall time:       2.7  microseconds         Iteration count:   256
Test description:
 Time to set a boolean flag using an "if" test
 a local and a global integer are compared
 compare this test with F000001
```

B.f. Loop Overhead

```
Test name:   L000001                        Class name:  Iteration
CPU time:        2.0  microseconds
Wall time:       2.0  microseconds         Iteration count:   2
Test description:
Simple "for" loop time
for I in 1 .. 100 loop
time reported is for once through loop


Test name:   L000002                        Class name:  Iteration
CPU time:        2.5  microseconds
Wall time:       2.5  microseconds         Iteration count:   2
Test description:
Simple "while" loop time
while I <= 100 loop
time reported is for once through loop


Test name:   L000003                        Class name:  Iteration
CPU time:        2.0  microseconds
Wall time:       2.0  microseconds         Iteration count:   2
Test description:
Simple "exit" loop time
 loop I:=I+1; exit when I>100; end loop;
time reported is for once through loop
```

B.g. Procedure Calls

There is no P000008 or P000009 test in the PIWG 8/31/86 suite.

```
Test name:   P000001                        Class name:  Procedure
CPU time:        0.4  microseconds
Wall time:       0.4  microseconds         Iteration count:   256
Test description:
 Procedure call and return time (may be zero if automatic inlining)
 procedure is local
 no parameters


Test name:   P000002                        Class name:  Procedure
CPU time:       54.7  microseconds
Wall time:      55.5  microseconds         Iteration count:   128
Test description:
 Procedure call and return time
 procedure is local, no parameters
 when procedure is not inlinable


Test name:   P000003                        Class name:  Procedure
CPU time:       42.2  microseconds
Wall time:      42.2  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 the procedure is in a separately compiled package
 compare to P000002


Test name:   P000004                        Class name:  Procedure
CPU time:        0.0  microseconds
Wall time:       0.0  microseconds         Iteration count:   256
Test description:
 Procedure call and return time measurement
 procedure is in a separately compiled package
 pragma INLINE used
 compare to P000001


Test name:   P000005                        Class name:  Procedure
CPU time:       44.5  microseconds
Wall time:      44.5  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 procedure is in a separately compiled package
 one parameter, in INTEGER
```

```
Test name:   P000006                        Class name:  Procedure
CPU time:       48.4  microseconds
Wall time:      48.4  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 procedure is in a separately compiled package
 one parameter, out INTEGER

Test name:   P000007                        Class name:  Procedure
CPU time:       51.6  microseconds
Wall time:      51.6  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 procedure is in a separately compiled package
 one parameter, in out INTEGER

Test name:   P000010                        Class name:  Procedure
CPU time:       74.2  microseconds
Wall time:      74.2  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER

Test name:   P000011                        Class name:  Procedure
CPU time:      106.2  microseconds
Wall time:     106.2  microseconds         Iteration count:   64
Test description:
 Procedure call and return time measurement
 compare to P000005, P000010
 20 parameters, in INTEGER

Test name:   P000012                        Class name:  Procedure
CPU time:       65.6  microseconds
Wall time:      65.6  microseconds         Iteration count:   128
Test description:
 Procedure call and return time measurement
 Compare with P000010 (discrete vs composite parameters )
 10 parameters, in MY_RECORD a three component record

Test name:   P000013                        Class name:  Procedure
CPU time:       93.8  microseconds
Wall time:      93.8  microseconds         Iteration count:   64
Test description:
 Procedure call and return time measurement
 twenty composite "in" parameters
 package body is compiled after the spec is used
```

B.h. Task Rendezvous

```
Test name:   T000001                        Class name:  Tasking
CPU time:     1662.5  microseconds
Wall time:    1662.5  microseconds         Iteration count:   8
Test description:
 Minimum rendezvous, entry call and return time
 one task, one entry, task inside procedure
 no select
```

```
Test name:   T000002                        Class name:  Tasking
CPU time:     1637.5  microseconds
Wall time:    1650.0  microseconds         Iteration count:   8
Test description:
 Task entry call and return time measured
 one task active, one entry in task, task in a package
 no select statement
```

```
Test name:   T000003                        Class name:  Tasking
CPU time:     1675.0  microseconds
Wall time:    1675.0  microseconds         Iteration count:   4
Test description:
 Task entry call and return time measured
 two tasks active, one entry per task, tasks in a package
 no select statement
```

```
Test name:   T000004                        Class name:  Tasking
CPU time:     1837.5  microseconds
Wall time:    1837.5  microseconds         Iteration count:   4
Test description:
 Task entry call and return time measured
 one task active, two entries, tasks in a package
 using select statement
```

```
Test name:   T000005                        Class name:  Tasking
CPU time:     1689.9  microseconds
Wall time:    1689.9  microseconds         Iteration count:   1
Test description:
 Task entry call and return time measured
 ten tasks active, one entry per task, tasks in a package
 no select statement
```

```
Test name:   T000006                        Class name:  Tasking
CPU time:     2429.9  microseconds
Wall time:    2419.9  microseconds         Iteration count:   1
Test description:
 Task entry call and return time measurement
 one task with ten entries, task in a package
 one select statement, compare to T000005


Test name:   T000007                        Class name:  Tasking
CPU time:     1612.5  microseconds
Wall time:    1600.0  microseconds         Iteration count:   8
Test description:
 Minimum rendezvous, entry call and return time
 one task one entry
 no select
```

Table of Contents