②

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Evaluation of the ACEC Benchmark Suite for Real-Time Applications**

CIN: **C02 092LY 0003 00**

23 July 1990

REPORT DOCUMENTATION PAGE

Form Approved OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 23 Jul 90	3. REPORT TYPE AND DATES COVERED Final Report

4. TITLE AND SUBTITLE Evaluation of the ACEC Benchmark Suite for Real-Time Applications	5. FUNDING NUMBERS DAAB07-87-D-B008 D.O. 2078
6. AUTHOR(S) Arvind Goel	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Unixpios Inc. 16 Birch Lane Colts Neck, NJ 07722	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army HQ CECOM Center for Software Engineering Fort Monmouth, NJ 07703-5000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT STATEMENT A Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Ada Compiler Evaluation Capability (ACEC) Version 1.0 benchmark suite was analyzed with respect to its measuring of Ada real-time features such as tasking, memory management, input/output, scheduling and delay statement, Chapter 13 features, pragmas, interrupt handling, subprogram overhead, numeric computations etc. For most of the features that were analyzed, additional benchmarks were proposed. The ACEC benchmarks were run on two Ada compilers (the HP Ada compiler self-hosted on HP 9000/350 and the Verdix Ada cross compiler hosted on the Sun 3/60 targeted to a Motorola 68020) and the results are listed.

14. SUBJECT TERMS Ada, ACEC, real-time, benchmarks	15. NUMBER OF PAGES 79
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

91 2 11 003

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-18
299-01

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	Contract	PR	Project
G	Grant	TA	Task
PE	Program Element	WU	Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	See authorities.
NASA	See Handbook NHB 2200.2.
NTIS	Leave blank.

Block 12b. Distribution Code.

DOD	DOD - Leave blank.
DOE	DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	NASA - Leave blank.
NTIS	NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Evaluation of the ACEC

Benchmark Suite for Real-Time Applications

Prepared For:

U.S. Army CECOM
Advanced Software Technology
Center for Software Engineering
Fort Monmouth, NJ 07703-5000

Prepared By:

Unixpros Inc.
16 Birch Lane
Colts Neck, NJ 07722

July 10, 1990.

Evaluation of the ACEC

Benchmark Suite for Real-time Applications

Abstract: This technical report has been developed by the Center for Software Engineering, US Army CECOM to evaluate the Ada Compiler Evaluation Capability (ACEC) Version 1.0 benchmark suite for measuring the performance of Ada compilers meant for programming real-time systems. The ACEC benchmarks have been analyzed extensively with respect to their measuring of Ada real-time features such as tasking, memory management, input/output, scheduling and delay statement, Chapter 13 features, pragmas, interrupt handling, subprogram overhead, numeric computations etc. For each of the features that have been analyzed, additional benchmarks have been proposed. Finally, the ACEC benchmarks that measure Ada features important for programming real-time systems have been run on two Ada compilers, namely the HP Ada compiler self-hosted on HP 900/350 and the Verdix Ada compiler hosted on the Sun 3/60 and targeted to a Motorola 68020 bare machine and their results listed.

CONTENTS

LIST OF TABLES

1. Introduction

This technical report has been developed by the Center for Software Engineering, US Army CECOM to evaluate the Ada Compiler Evaluation Capability (ACEC) benchmark suite (developed by Boeing Aerospace under contract to the US Air Force [1], [2]) for measuring the performance of Ada compilers meant for programming real-time systems. In this report, the ACEC benchmarks have been analyzed for their coverage and suitability in measuring ihe runtime performance of Ada features that are important for real-time applications. The emphasis of the ACEC benchmarks is on measuring the runtime performance of Ada compilers, although compilation issues have also been addressed.

The principal goal of Ada is to provide a language supporting modern software engineering principles in the design and development of real-time systems. To design and implement real-time systems, it is essential that the performance as well as implementation characteristics of an Ada compiler system meet the requirements of a real-time application. Many current Ada implementations do not allow the development of reliable embedded systems software without sacrificing productivity and quality. One of the reasons for the above is that real-time programmers have no control on the design and implementation of the Ada Runtime System (RTS) except that the RTS satisfy the requirements listed in the Ada Language Reference Manual (LRM). Due to the effect on program efficiency and reliability of the various runtime implementation options, simply adopting a compiler that implements the language as defined in the LRM is insufficient for real-time systems. Benchmarks are needed to determine the performance as well as implementation strategies of various Ada language and runtime features in order to assess a compiler's suitability for programming real-time systems.

1.1 Objective

The CECOM Center for Software Engineering, US Army, Fort Monmouth has been involved with developing benchmarks for Ada language and runtime system features considered important for programming real-time applications. The first step in this effort involved the identification of Ada features of interest for real-time systems. The real-time systems analyzed were the COMINT/ELINT (Communication/Electronic Intelligence) class of IEW (Intelligence/Electronic Warfare) systems supported by the US Army. A side result of this effort was the description of a composite benchmark for COMINT/ELINT class of IEW systems [3].

The next step involved the development of real-time benchmarks that measure the Ada features identified in the previous effort. Benchmarks were developed that a) measure the performance of Ada individual features, b) determine Ada runtime system implementation dependencies, and c) test algorithms used in programming real-time systems [4].

As part of this ongoing effort, existing benchmark suites were also analyzed to determine their suitability for evaluating Ada compiler systems for real-time applications. The benchmark suites that were analyzed included

- *PIWG Benchmarks:* developed by the ACM Performance Issues Working Group [5].
- *University of Michigan Benchmarks:* developed at the University of Michigan [6].
- *Ada Compiler Evaluation Capability:* developed by Boeing Aerospace for the US Air Force [1], [2].

This report presents the results of the analysis of ACEC benchmarks for measuring the runtime performance of Ada features important for programming real-time applications. The benchmarks have been analyzed with respect to:

- support for Ada features important for programming real-time applications
- interpretation of the results produced by running the benchmarks
- and portability of the ACEC benchmarks.

The results of running a few selected ACEC benchmarks (that measure real-time Ada features) on two Ada compilers, namely the HP Ada Ada compiler which is self-hosted on a HP 9000/350 computer running HP-UX and a Verdix cross-compiler hosted on a Sun 3/60 and targeted to a Motorola 68020 bare machine are also presented.

1.2 Report Layout

This report is divided in the following sections:

Section 2 presents a brief description of the ACEC benchmarks.

Section 3 describes the typical requirements of real-time systems and correlates them with Ada features that address those requirements. It also discusses the criteria for analysis of the ACEC benchmarks.

In Section 4, the ACEC benchmarks are analyzed with respect to the Ada features identified in Section 3. Section 4 also comments on the portability of the benchmarks.

Finally, Section 5 concludes with some thoughts about the ACEC benchmarks.

Appendix A consists of a list of tables that describe the ACEC real-time benchmarks. It also lists the results of running the ACEC real-time benchmarks on the HP and Verdix Ada compiler systems.

Appendix B comments on the usefulness of the ACEC benchmarks in evaluating Ada compilers from a real-time perspective.

2. Ada Compiler Evaluation Capability (ACEC) Suite

The ACEC benchmarks were developed by Boeing Aerospace Company under contract to the US Air Force. The ACEC is organized as a set of essentially independent test problems and analysis tools. The major emphasis of the ACEC is on execution performance. To a lesser degree, the ACEC will also test for compilation speed, existence of language features, and capacity. The tests are designed to:

- Produce quantitative results, rather than subjective evaluations
- Be as portable as possible
- Require minimal operator interaction
- Be comparable between systems, so that a problem run on one system can be directly compared with that problem run on another target.

The ACEC does not address issues such as: cost, diagnostics and error handling, adaptability to a special environment, presence of support tools, and target processors such as vector processors, VLIW machine architectures, RISC processors, multicomputers.

The ACEC contains a large number of test problems (~ 1000). Most individual problems are fairly small. Many address one language feature or present an example which is particularly well suited to the application of a specific optimization technique.

The primary focus of the ACEC is on comparing performance data between different compilation systems rather than on studying the results of one particular system. The analysis tool MEDIAN computes overall relative performance factors between systems and isolates test problems where any individual system is much slower (or faster) than expected, relative to the average performance of all systems on that problem and the average performance of the problems on all systems. ACEC users can review the MEDIAN report to isolate the strong and weak points of an implementation by looking for common threads among test problems which report exceptional performance data. The ACEC comparative analysis programs compare performance data between systems and identify the test problems which show statistically unusual results. The results of some test problems are of independent interest - such as rendezvous times, exception propagation time, and procedure call time.

The ACEC addresses the following:

1. Execution Time Efficiency
2. Code Size Efficiency
3. Compile Time Efficiency (to a much lesser degree).

2.1 Execution Time Efficiency

These benchmarks address the execution speed of various Ada features as well as those aspects of an Ada Runtime System which traditionally have been the province of operating systems. These benchmarks have been subdivided into the following major categories:

1. Benchmarks that deal with Ada Runtime System features
2. Benchmarks that measure individual Ada language features
3. Benchmarks that deal with performance under load
4. Benchmarks that deal with tradeoffs between performance of different features
5. Benchmarks that deal with optimization issues.
6. Application Profile benchmarks that are further subdivided as follows:
   - Classical Benchmark Programs (e.g. Whetstone, Dhrystone)
   - Ada in Practice

2.1.1 ACEC Ada Runtime System Benchmarks

These benchmarks address Ada Runtime System Issues that traditionally have been the domain of operating systems as well as determining runtime implementation dependencies. The benchmarks address Ada RTS issues such as:

1. *Tasking:* Tasking benchmarks can be further subdivided as follows:
   a. Task Activation/Termination
   b. Task Synchronization
   c. Exceptions Raised During Rendezvous
   d. Abort Statement
   e. Tasking Runtime Implementation Dependencies
   f. Tasking Optimizations
2. *Memory Management:* Memory management benchmarks have been subdivided into two areas:
   a. Memory Allocation Timing Benchmarks: These benchmarks are mainly tests that determine timing information about memory allocation/deallocation.
   b. Memory Allocation/Deallocation Benchmarks: These benchmarks determine the way storage allocation/deallocation is implemented for a particular Ada compiler system.

3. *Exception Handling:* These benchmarks measure the time to raise, propagate and handle exceptions.
4. *Input/Output:* These benchmarks measure time for input/output operations for TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO. Although many embedded targets do not support file systems, embedded applications may make intensive use of file systems and the performance of I/O operations is critical to their application performance.
5. *CLOCK Function:* These benchmarks determine the overhead due to the CLOCK and SECONDS function.
6. *Chapter 13 benchmarks:* These benchmarks measure the performance of various Chapter 13 features such as Pragma PACK, SIZE Representation Clause, Record representation clause, and Unchecked Conversion.
7. *Scheduling and Delay Statement:* These benchmarks determine the scheduling algorithms and the impact of the delay statement.
8. *Pragmas:* There are certain predefined pragmas which are expected to have an impact on the execution time and space of a program. These include: Pragmas CONTROLLED, INLINE, OPTIMIZE, PACK, PRIORITY, SHARED, and SUPPRESS. There are test problems which explore the performance effects of specifying the above pragmas.

2.1.2 ACEC Individual Language Feature Benchmarks

There are test problems for all major Ada language features. The test suite contains sets of test problems which present constructions in different contexts. The results will demonstrate the range of performance associated with a language feature. Benchmarks that measure individual language features are divided into the following categories:

1. *Record object manipulation*
2. *Array object manipulation*
3. *Integer Computations*
4. *Floating point computations*
5. *Fixed point computations*
6. *Loop operations*
7. *Constraint checking*
8. *Type conversions from one type to another*
9. *Mathematical functions*
10. *Subprogram Overhead (including Generics)*

2.1.3 Performance Under Load

There are some language constructions which display nonuniform performance. The more of them in a program, the slower the average performance. These are tests that determine the performance of such language constructions under different loading scenarios. Examples include task loading, levels of nesting, parameter variation, and declarations.

2.1.4 Tradeoffs

In many areas of the language, it is possible to speed up the performance of one feature at the cost of slowing another down. Areas that have been covered include design issues, and context variation.

2.1.5 Optimizations

Specific optimization test problems include examples where it is easy and where it is more difficult to determine that the optimization is applicable. There are some test problems which perform the same basic operations, but have a modification which either performs the intended optimization in the source text, or precludes the application of the optimization. Optimizations that have been addressed include:

1. Common Subexpression Elimination
2. Folding
3. Loop Invariant Motion
4. Strength Reduction
5. Dead Code Elimination
6. Register Allocation
7. Loop Interchange
8. Loop fusion
9. Test Merging
10. Boolean Expression Optimization
11. Algebraic Simplification
12. Order of Expression Evaluation
13. Jump tracing
14. Unreachable Code Elimination

15. Use of Machine Idioms

2.1.6 Application Profile Tests

The ACEC also includes examples from actual application code. This code contains test problems representative of how Ada is being used in practice. These are example test problems drawn from Ada programs extracted from projects. Examples include Avionics Applications, Electronic Warfare Application feasibility study code, and Radar Application code.

Application profile tests have been subdivided as follows:

1. *Classical Tests:* The ACEC test suite contains classical benchmark programs coded in Ada. Classical tests include: Ackermann's function, Kalman filter, Autocorrelation program, Quicksort and variation of quicksorts, Mergesort, Dhrystone and Whetstone benchmarks, and Gamm measure benchmark.
2. *Ada in Practice:* These are example test problems drawn from Ada programs extracted from projects. They represent typical usage of Ada.

2.2 Code Size Efficiency

The memory size of programs is an important attribute in many mission critical applications. On embedded systems, memory is often a limited resource. On some target processors such as the MIL-STD-1750A, while physical memory may be available, maintaining addressability is critical and a small code expansion rate can help system design by reducing the need to switch memory states. There are two size measurements of most interest to Ada projects: the amount of space generated inline to translate each statement (Code Expansion Size), and the amount of space occupied by the RTS (Runtime System Size).

Code Expansion Size: The code expansion size is measured in the timing loop. It is the space in bits, between the beginning and end of each test problem. In this report, the code expansion size of problems have been considered in light of their ability to help interpret results.

Runtime System Size: The size of the Runtime System is an important parameter to many projects. Space taken by the RTS is not available for use by application code, so a small RTS will permit larger applications to be developed.

2.3 Compile Time Efficiency

The times to compile the compilation units are collected and analyzed. The benchmarks were developed to measure execution time performance aspects, and do not necessarily represent a set of compilation units which will expose all the relevant compilation time variables. However, they do represent a set of programs which will exercise a compiler and observing the compile time of these programs can give insight to the overall compilation rates.

2.4 ACEC Summary

The philosophy of the ACEC is that end users will not have to examine in detail each individual test problem. Rather, they should run the test suite and let the analysis tools isolate problems where a system does unusually well or unusually poorly. These problems can then be examined in more detail to try to determine what characteristics of the problem are responsible for the unusual behavior.

More information about the ACEC benchmarks can be obtained from references [1] and [2] as well as from:

Raymond Szymanski
WRDC/AAAF-3
Wright-Patterson AFB
Ohio 45433-6543
(513) 255-3947

3. Real-time Systems and Ada Benchmarking

Before jumping into the ana.,/sis of ACEC benchmarks, it is important to understand the typical software requirements of real-time applications, how Ada addresses those requirements and the issues involved in benchmarking Ada features that address these requirements.

3.1 Requirements Of Real-time Systems

For a programming language to be used effectively to program real-time embedded systems, it should be able to support the following characteristics (for more details see reference [4]):

- *Real-time Preemptive Scheduling*
- *Concurrency, Inter-task and Intra-task communication*
- *Time Abstraction*
- *Interaction with Real World*
- *Input/Output*
- *Resource Utilization*
- *Numeric Computations*
- *Fault Tolerance*
- *Event-driven Reconfiguration*
- *Reliability*

3.2 Ada and Real-time Requirements

Programmers generally have no control on the design and implementation of the Ada runtime system except that it satisfy the requirements listed in the LRM. Table 1 lists the Ada features that support real-time requirements.

TABLE 1. Real-time Requirements and Ada

Real-time Requirements	Ada Feature
Real-time, Preemptive Scheduling	Ada Runtime System, Priority Mechanism,
Control Over Timing	Delay Statement, CLOCK function, TYPE TIME and DURATION
Concurrency, Inter-task and Intra-task Communication	Ada tasking mechanism, Rendezvous, Procedure Calls
Interaction with real world	Address clause binding entry to an interrupt
Input/Output	TEXT_IO, SEQUENTIAL_IO, DIRECT_IO, Chapter 13 features
Numeric Computations	Math library, Chapter 13 features
Resource Utilization	Delay Statement, Memory Management features, Pragmas
Fault Tolerance	Exception Handling Mechanism
Asynchronous Change in Control	Partially addressed via abort statement
Distributed Architectures	Not explicitly addressed by a specific feature

From Table 1, it is clear that a real-time benchmarking suite should address the following real-time Ada features:

1. *Tasking*
2. *Memory management*
3. *Exceptions*
4. *Input/Output*
5. *Clock Function*
6. *Scheduling and Delay Statement*
7. *Chapter 13 Benchmarks*
8. *Interrupt Handling*
9. *Pragmas*
10. *Subprogram Overhead*
11. *Numeric Computations*

3.3 Ada Benchmarking

Ada benchmarking can be approached in 4 ways:

1. Benchmarks that measure execution speed of individual features of the language.
2. Benchmarks that determine implementation dependent attributes.
3. Benchmarks that measure the performance of commonly used real-time Ada paradigms (that may be programmed using macro constructs [4]).
4. Composite benchmarks which include representative code from real-time applications.

A detailed description of benchmarking approaches is presented in the report titled "Real-time Performance Benchmarks For Ada" [4].

4. Analysis of the ACEC Benchmarks

The ACEC benchmarks have been analyzed with respect to the following characteristics:

1. *Features measured by the Benchmarks*

   The ACEC suite has been analyzed with respect to its ability to measure the performance and to determine the implementation characteristics of Ada features that are important for programming real-time systems. The ACEC tests are examined with respect to each of the following real-time features [4]:

   - Tasking
   - Memory management
   - Exceptions
   - Input/Output
   - Clock Function
   - Scheduling and Delay Statement
   - Chapter 13 Benchmarks
   - Interrupt Handling
   - Pragmas
   - Subprogram Overhead
   - Numeric Computations

   In addition to their measurement of individual Ada features, the ACEC suite has also been evaluated with respect to its implementation of real-time paradigms and composite benchmarks.

2. *Information provided for interpretation of the results*

   Running the ACEC benchmarks produces a set of numbers which have to be interpreted. It is important that the benchmarking suite provide sufficient information about interpreting the results.

3. *Portability*

   The benchmarks should be portable and executable on any Ada compiler system with minimum modifications. There are some benchmarks (like interrupt handling) that may not be portable and depend on the hardware being tested. The ACEC suite has been analyzed with respect to its ease of portability to various Ada compiler systems.

The ACEC evaluation format is as follows:

- For each feature, the ACEC benchmarks which address that feature have been identified and their description presented in tables listed in Appendix A. Also listed are the results of running the ACEC It also presents the results of running the ACEC real-time benchmarks on the following Ada compilers: HP-Ada Compiler (Releases 3.25 and 4.35) running on HP 9000/350 machine under HP-UX Release 6.2; and Verdix Ada Compiler (Release 5.41) hosted on a Sun 3/60 and targeted to a Motorola 68020 bare machine.
- Then, comments are presented on those set of benchmarks. The comments address two major areas, namely any deficiencies in a) the benchmarks themselves and b) additional information that is not provided by the ACEC in interpretation of the results produced by running those benchmarks.
- Finally, additional benchmarks not covered in the ACEC are listed when appropriate for the feature analyzed.

4.1 Tasking

For the purposes of this discussion, the ACEC benchmarks have been analyzed with respect to their measurement of the following aspects of tasking:

- Task Activation/Termination
- Task Synchronization
- Exceptions Raised During Rendezvous
- Abort Statement
- Tasking Priorities
- Miscellaneous Tasking Benchmarks
- Tasking Optimizations

4.1.1 Task Activation/Termination

Task Activation/Termination is an important benchmark for real-time systems. Task elaboration, activation and termination are almost always suspect operations in real-time programming and programmers often allocate tasks statically to avoid runtime execution time.

Table 2 lists the ACEC benchmarks for measuring task activation/termination timings along with the results of running these benchmarks on the HP and Verdix Ada compilers. The task to be activated can either be an object of a task type or can be activated using the **new** allocator. The difference in the times provided by these tests give some insight into the relative efficiency of the two types of task activation.

Comments: Observations about the ACEC task activation/termination benchmarks are:

1. The tasks whose activation/termination times are being measured are very simple tasks which have a null statement inside the task body. This is not a very realistic scenario as it is quite possible that many compilers realizing that this task does nothing may optimize it away and the measurements obtained may not be correct. The task body should do something meaningful such as call a subprogram that performs some meaningful calculations.

2. The time to elaborate, activate and terminate a task is measured as one value. The individual components of the measurements are too quick to measure with the available CLOCK resolution.

3. An important criteria for tasking benchmarks is the STORAGE_SIZE used by the tasks that are elaborated. Some implementations may implicitly deallocate the task storage space on return from a procedure or on exit from a block statement (when the task object is declared in that procedure or block statement). If task space is implicitly deallocated, the number of iterations can be increased to get greater accuracy for task activation/termination measurement. So if task space is not deallocated on return from a procedure or block statement, TASK_TYPE'STORAGE_SIZE can be changed such that the number of iterations can be increased (thus increasing the accuracy of the measurement).

Additional Task Activation/Termination Benchmarks:

1. More task activation/termination benchmarks are needed to determine if a real-time programmer can declare tasks for time-critical modules in a) a block statement or b) within other tasks.

   Benchmark: Measure task activation and termination time (without the new operator) where

   - *Task type is declared in the main program and task object is declared in a block statement in the main program.*

   - *Task type and task object are declared in another task which is declared in the main program.*

2. Task activation/termination times may degrade as the number of active tasks in the system increases. This is a more realistic scenario for a real-time system as generally there are existing tasks when new tasks are activated. As more and more tasks are created, task activation time may increase due to the possible increase in storage allocation time.

   Benchmark: Measure the affect on task activation/termination times as the number of existing active tasks keeps on increasing.
   Number of existing active tasks in the system could vary from 1 to 20.

3. During the execution of an Ada program, a low priority task spawns a task. While the activation of this spawned task is occurring, if a high priority task becomes ready to execute, it may remain suspended until the completion of the low priority task activation.

   Benchmark: Determine if a low priority task activation could result in a very long suspension of a high priority task.

4.1.2 Task Synchronization

In Ada, tasks communicate with each other via the rendezvous mechanism. Rendezvous are effectively similar to procedure calls, yet they are much more complex to implement, and therefore create a tremendous amount of overhead for the runtime system. Because of the timing constraints in a real-time system, it is essential that the rendezvous mechanism be as efficient as possible.

The ACEC suite has a comprehensive set of task synchronization benchmarks. These benchmarks are divided in the following logical areas.

4.1.2.1 Time For a Simple Rendezvous

These benchmarks measure the time for a simple rendezvous and no parameters are passed during the rendezvous. Time is measured to complete a rendezvous between a task and a procedure with no additional load present. This method, then gives a lower bound on rendezvous time, because no extraneous units of execution are competing for the CPU. Table 3 lists the simple rendezvous benchmarks and Table 4 lists the benchmarks that determine rendezvous performance with varying number of tasks. The results of running these benchmarks on the HP and Verdix Ada compilers are also listed.

Comments: Comments about ACEC simple rendezvous benchmarks are:

1. The ACEC benchmarks measure simple rendezvous timings for rendezvous between equal priority tasks, as well as rendezvous between tasks of different priorities. Two context switches are required for rendezvous between tasks of different priorities as opposed to a single context switch for rendezvous between tasks of same priority. So any deviation from the expected results points to a bad compiler implementation.

2. Rendezvous times with tasks in subunits as well as tasks in separate packages should be the same as if those tasks were in the main program. A task being in a subunit or a separate package should affect compilation times and not execution times.

4.1.2.2 Select Statement With Else Alternative

These benchmarks measure the time it takes to execute a select statement with an else alternative under various scenarios. The ELSE alternative is always executed. Table 5 lists these benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. All of these benchmarks have a select statement with an else alternative (which is always executed). Hence, execution of these benchmarks will enable real-time programmers to determine if a compiler optimizes away the entry call or accept statement and directly executes the else alternative.

4.1.2.3 Rendezvous Calls with Conditional Selects

These benchmarks have conditional select statements with various scenarios. The tests are listed in Table 6 along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. Benchmarks (task16) that contain delay statements with a negative argument do not serve any useful purpose for real-time programmers, as negative delay statements are never used in real-time systems.
2. There is no information provided on interpretation of the results produced by executing benchmarks task15, task16, task17, task21, and task22. Upon further analysis, it is determined that these benchmarks do not provide any useful information in evaluating an Ada compiler.

4.1.2.4 Selective Wait

These benchmarks test various scenarios with the selective wait statement. Table 7 lists the selective wait benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. Benchmark (task34) measures the time to check the entry queue for a accept statement. Depending on the compiler implementation, time to check the entry queue could force missed deadlines in real-time systems.
2. Benchmarks (task35) that contain delay statements with a 0.0 argument do not serve any useful purpose for real-time programmers, as delay statements with 0.0 argument are never used in real-time systems.

3. In benchmark task59, all delay alternatives are negative in the selective wait. As pointed out before, negative delay alternatives do not provide any useful information in Ada compiler evaluation for real-time programming.

Additional Task Synchronization Benchmarks:

1. For some compilers, the more the number of entries in a select statement, the more time it takes to rendezvous with any entry in the select statement. For some implementations, time for a rendezvous may also be affected by the position of the accept alternative in the select statement.

   Benchmark: Measure the effect on rendezvous time as the number of accept alternatives in a select statement increases.

   Main program calls first (middle, last) entrv i[illegible] [illegible] select statement in another task as the number of accept statements increases from 2 to 10 to 20.

2. Rendezvous time may depend on the number of open guard statements. For some implementations, rendezvous time may depend on the number of guards in the select statement and the position of the accept in the select statement.

   Benchmark: Measure the effect of guards on rendezvous time, where the main program calls an entry in another task as the number of accept alternatives in the select statement increases.

3. Rendezvous time may depend on the size and type of the passed parameters which may involve both the task stacks or the allocation of a separate area for passing large structures. Increasing rendezvous times for array parameters as the size of the array increases implies that the implementation uses pass by copy [illegible]stead of pass by reference.

   Benchmark: Measure the time required for a complex rendezvous, where a procedure in the main program calls an entry in another task with different type, number and mode of the parameters.

   The types of the parameters include a) integer arrays (size 1 to 1000 to 10000), and b) 1 to 100 integers. The mode of the parameters passed is either *out* or *in out*.

4. Fairness of select-alternative is a particular aspect of scheduling fairness. If a task reaches a selective wait and there is an entry call waiting at more than one open alternative, or if a task is waiting at a selective wait and more than one open accept or delay alternative becomes eligible for selection at the same time, an alternative is selected according to criteria that are not specified in the LRM.

Benchmark: Determine algorithm used when choosing among branches of a selective wait statement.

5. The order in which an Ada compiler system chooses to evaluate the guard conditions in a select statement is implementation dependent. Real-time programmers may need to know the order in which the guard conditions are evaluated.

   Benchmark: Determine the order of evaluation for guard conditions in a selective wait.

4.1.3 Exceptions During a Rendezvous

If an exception is raised within a rendezvous, it is propagated to the task containing the accept as well as to the calling task. This is the most complex form of exception handling since the exception is handled in both the task containing the accept and the calling task. For real-time systems, it is important to measure the time it takes to handle exceptions raised during a rendezvous. Table 8 lists the ACEC benchmarks for exceptions raised during a rendezvous along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. ACEC benchmarks for handling exceptions within a rendezvous are adequate and do not require any additional benchmarks.

4.1.4 Abort Statement

Quick restarts of tasks are required in a number of real-time embedded systems. Ada model of concurrency does not provide an abstraction where a task may be asynchronously notified that it must change its current execution state. One way to implement asynchronous change in control is to abort the task and then replace it with a new one. Table 9 lists the tests for abort statement along with the results of running these benchmarks on the HP and Verdix Ada compilers. These tests measure timings to abort tasks under various scenarios.

Comments: None.

Additional Abort Statement Benchmarks:

More task abortion benchmarks are needed as follows:

1. In real-time systems, tasks may have to be aborted in a certain sequence. The semantics of the abort statement do not guarantee immediate completion of the named task. Completion must happen no later than when the task reaches a

synchronization point.

Benchmark: Determine order of evaluation of tasks named in an abort statement.

2. When a task has been aborted, it may become completed at any point from the time the abort statement is executed until its next synchronization point. Depending on when an implementation actually causes the task to complete the results of an aborted task may be different. Suppose a task is updating a variable that is visible to other tasks, prior to a synchronization point. If the task is aborted just prior to the update, it may leave the variable unchanged if it becomes completed immediately, or it may update the variable and then becomes completed at the synchronization point. This could affect the results of the whole program.

   Benchmark: Determine the results if a task is aborted while updating a variable ?

4.1.5 Task Priorities

The ACEC suite has several tests that use Pragma PRIORITY to determine rendezvous timings under different scenarios. These tests have already been discussed in Section 4.3.2 under Task Synchronization. However, additional benchmarks are needed for tasking priorities.

Additional Tasking Priority Benchmarks:

1. Programmers may need to know the default priority of the main program and other tasks in order to design usable embedded systems.

   Benchmark: Determine priority of tasks (and of the main program) that have no defined priority.

2. If two tasks without explicit priorities conduct a rendezvous, and if the priority given to the rendezvous is higher than a task with an explicit priority, the Ada program may perform in an unpredictable manner.

   Benchmark: Determine priority of a rendezvous between two tasks without explicit priorities.

4.1.6 Miscellaneous Tasking Benchmarks

There are some tasking benchmarks that do not fall under any of the above defined categories.

Table 10 lists the miscellaneous tasking benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. Benchmark (task48) is not very useful as the timing of interest is the time taken to invoke the interrupt handler when an actual hardware interrupt is received as opposed to calling an entry tied to an interrupt directly.

Additional Miscellaneous Benchmarks:

1. A group of tasks (children of the same parent) can terminate by using the terminate option of the select statement. If the overhead due to the terminate option is high, then this option should not be used (especially if the selective wait is inside a loop).

   Benchmark: Measure the cost of using the terminate option in a select statement.

2. In many real-time embedded systems where space is at a premium it may be desirable that task space be deallocated when that task terminates.

   Benchmark: Determine if task space is deallocated on return from a procedure when a task that has been allocated via the new operator in that procedure terminates.

3. It might be impossible for a runtime system to deallocate the task storage space after termination. This is because the access value might have been copied and an object might still be referencing the terminated task's task control block.

   Benchmark: Determine if tasks that are allocated dynamically by the execution of an new allocator do not have their space reclaimed upon termination when access type is declared in a library unit or outermost scope.

4. When several tasks are activated in parallel, the order of their elaboration may affect program execution.

   Benchmark: Determine the order of elaboration when several tasks that are declared in the same declarative region are activated in parallel.

5. The activation of tasks proceeds in parallel. Correct execution of a program may depend on a task continuing execution after its activation is completed but before all other tasks activated in parallel have completed their respective activations.

   Benchmark: Determine if a task, following its activation but prior to the completion of activation of tasks declared in the same declarative part, continue execution.

6. The LRM does not define when STORAGE_ERROR must be raised should a task object exceed the storage allocation of its creator or master. The exception must be no later than task activation; however an implementation may choose to raise it earlier.

   Benchmark: Determine when exception is raised if the allocation of a task object raises STORAGE_ERROR.

7. For some real-time embedded applications, it is desirable that tasks declared in a library package do not terminate when the main program terminates. System designers may need to know this information.

   Benchmark: Determine if tasks declared in a library package terminate when the main program terminates.

4.1.7 Task Optimization

These benchmarks are designed to determine if certain tasking optimizations have been implemented by Ada compilers.

Table 11 lists the ACEC Habermann-Nassi tasking optimization benchmarks Table 12 lists the other tasking optimization benchmarks. The results of running these benchmarks on the HP and Verdix Ada compilers are also listed.

Comments:

1. As far as tasking optimizations are concerned, the ACEC suite is quite comprehensive.

4.2 Memory Management

Memory management benchmarks have been divided into two separate areas:

1. *Memory Allocation Timing Benchmarks:* These benchmarks are mainly tests that determine timing information about memory allocation/deallocation.
2. *Memory Allocation/Deallocation Benchmarks:* These benchmarks determine the way storage allocation/deallocation is implemented for a particular Ada compiler system.

4.2.1 Memory Allocation Timing Benchmarks

Since time and space are at a premium in real-time embedded systems, it is essential that the dynamic memory allocation and deallocation be as efficient as possible.

Real-time programmers need to know the maximum time to allocate and deallocate storage for a particular Ada compiler in order to ensure that performance requirements will be met for their application.

Table 13 lists the ACEC memory allocation timing benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. In benchmarks (ss22, ss23, and ss24), time to to allocate small one, two, and three dimensional arrays of float is measured. The sizes of the arrays are different thus limiting the usefulness of the results obtained by running these benchmarks. What is of interest is the time for allocating same size array as the number of dimensions increase.
2. Benchmark (ss25), which measures time to allocate a small dynamically bounded one array of float, does not provide any useful information as the timing has to be compared to allocation timings for same size and multiple dimension arrays.

Additional Memory Allocation Timing Benchmarks:

1. Tests for more Ada types are needed to determine their allocation overhead time. Times to allocate various numbers of types INTEGER and ENUMERATION have to be measured as well as the times to allocate various sizes of arrays, records, and STRINGs. The objective is to determine the allocation overhead involved and if there is any difference in the overhead based on the type of object allocated.

   Benchmark: Measure time for allocating storage known at compile time. Times to allocate various numbers of types INTEGER and ENUMERATION are measured as well as the times to allocate various sizes of arrays, records, and STRINGs.

2. More tests are needed to determine if allocation time is dependent on size (in the composite-type object case). Also, based on these timing measurements real-time programmers can decide whether to use the new allocator for object elaboration or to declare the object as in the fixed length case.

   Benchmark: Memory Allocation via the New Allocator. Allocation time of objects of type INTEGER, and ENUMERATION as well as composite type objects of various sizes are measured.

   In these tests, the objects that have been allocated via the new allocator have also been freed via Unchecked_Deallocation before exiting the scope in which the object was allocated.

3. If memory is allocated in a loop via the new allocator and the memory that is allocated is not freed via Unchecked_Deallocation, then the time required for dynamic memory allocation can be affected as more space is allocated.

   Benchmark: Determine the effect on time required for dynamic memory allocation when memory is continuously allocated without being freed via Unchecked_Deallocation in the scope where the memory was allocated.

4.2.2 Memory Allocation/Deallocation Benchmarks

It is important for real-time programmers to know if a particular compiler implementation

- deallocates nothing
- supports only UNCHECKED_DEALLOCATION
- deallocates all the storage for an access type when the scope of the access type is left
- detects inaccessible storage and automatically deallocates it (garbage collection).

Table 14 lists the ACEC memory allocation/deallocation benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. ACEC suite is very comprehensive in memory allocation/deallocation benchmarks.

4.3 Exceptions

Table 15 lists the ACEC exception handling timing benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: None.

Additional Exception Handling Benchmarks:

1. Exception handling times may degrade due to additional tasks present in the system. Benchmarks are needed that measure exception handling timings when multiple tasks are present in the system.

   Benchmark: Measure the effect of additional tasks in the system on exception handling times for all the exception handling benchmarks in the ACEC suite.

2. In many real-time systems, it is quite possible that intermediate operations during the calculation of a larger expression may exceed the system defined limits, although the final result may still be within bounds. Some implementations may raise an exception if the intermediate expression exceeds system defined limits.

   Benchmark: Determine if an implementation raises NUMERIC_ERROR on an intermediate operation when the larger expression can be correctly computed.

4.4 Input/Output

Input/Output benchmarks can be divided into the following categories:

- TEXT_IO benchmarks
- DIRECT_IO benchmarks
- SEQUENTIAL_IO benchmarks
- Asynchronous I/O benchmarks

4.4.1 TEXT_IO

Table 16 lists the ACEC TEXT_IO benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. Additional benchmarks for TEXT_IO are not needed.

4.4.2 DIRECT_IO

Table 17 lists the ACEC DIRECT_IO benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. Additional benchmarks for DIRECT_IO are not needed.

4.4.3 SEQUENTIAL_IO

Table 18 lists the ACEC SEQUENTIAL_IO benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. Additional benchmarks for SEQUENTIAL_IO are not needed.

4.4.4 Asynchronous I/O

One of the benefits of Ada's tasking techniques is the ability to implement true asynchronous I/O. By using Ada tasks to drive I/O controllers, only the task that requested the I/O must wait for completion before resuming execution, while other tasks within the application program can continue execution while I/O is being processed.

I/O blocking may not be tolerated in many systems. It effectively causes the entire Ada program to stop while an I/O is serviced. The effect is clearly most evident for interactive input but, for mission critical systems even physical disk I/O will cause unacceptable delays in the overall processing.

Comments:

1. The ACEC suite does not have benchmarks that address asynchronous I/O.

Additional Asynchronous I/O Benchmarks

1. I/O blocking to devices other than where clear delays are possible (such as a terminal or mailbox) can be very difficult to determine. In principle it is only necessary for non-blocking I/O to occur for physical I/O but when this actually happens is difficult to predict in many systems where complex device caching and buffering is automatically performed. The tests need to be performed for the following device types: interactive terminal and disk. For each facility and each of SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO the presence of system-wide blocking during prolonged processing should be recorded.

   Benchmark: Blocking on READ, GET, WRITE, PUT, CREATE, OPEN, RESET, CLOSE, and DELETE.

4.5 Clock Function

For programming real-time systems, the CLOCK function in the package CALENDAR is used extensively. Table 19 lists the ACEC CLOCK function tests along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. These tests are sufficient to test the CLOCK function overhead.

4.6 Scheduling, Preemption and Delay Statement

To allow execution to switch among tasks, the scheduler provided by the runtime system is entered at certain synchronization points in a program, and the scheduler decides at this point which task has to be executed. According to the LRM, an implementation is free to choose among tasks of equal priority or among tasks whose priority has not been defined. The minimum synchronization (a implementation may choose to have more) points at which the scheduler is invoked are the beginning and end of task activations and rendezvous. The pragma priority enables real-time embedded systems programmers to specify a higher priority for more important tasks. The priority is fixed at compile time (assuming that pragma priority is implemented). Hence, whenever a scheduling decision has to be made, the highest priority task receives control (task priorities are discussed in Section 4.1.5).

Table 20 lists the ACEC benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about these benchmarks are:

1. As mentioned in previous sections, benchmarks that measure timing for delay statement with negative (ss459) or 0.0 (delay1, delay8) argument do not serve any useful purpose for real-time systems.

Additional Scheduling, Preemption and Delay Statement Benchmarks:

Task preemption feature is very important for real-time systems. Preemption occurs for a variety of reasons each of which must be established. Linked with preemption is the scheduling algorithm used to determine which, of any number of candidates, can be processed. The test should determine, as far as possible, the conditions under which tasks are scheduled and, in particular, the order chosen where valid alternatives exist.

1. Expiration of delay statement may cause scheduling to take place and preempt the running task.

   Benchmark: Test whether delay expiration causes task preemption.

2. Some implementations may cause scheduling decisions to take place upon I/O completion. It should be ascertained if RTS system calls (such as opening of files, task creation, and rendezvous handling) are themselves preemptable. This is of great importance when dealing with multi-priority systems and especially where interrupts are possible.

   Benchmark: Establish whether I/O completion causes task preemption.

3. An external interrupt may also cause preemption.

   Benchmark: Establish whether external interrupts preempt running tasks.

4.7 Chapter 13 Benchmarks

Chapter 13 benchmarks are divided into the following categories:

1. Pragma PACK: This is considered here (as opposed to the section that deals with Pragmas) as the extent of packing performed by a compiler can be compared to other benchmarks that use the SIZE specification clause.
2. SIZE specification benchmarks
3. Record repres ..t..ion clause benchmarks
4. Attribute be chmarks
5. Unchecked_Conversion benchmarks

4.7.1 Pragma Pack

These set of benchmarks test the packing capabilities of a Ada compiler system by specifying Pragma Pack for various objects. These tests measure both time and space utilization. Some packing methods allocate a component so that it will span a storage unit boundary while some pack as densely as possible. The time to access a component which spans a storage unit is usually greater than when the component does not span a boundary. In addition to measuring the time in accessing packed objects, these test problems use the representation attribute X'SIZE to determine the actual bit size of the objects and compare this with the predetermined minimum possible bit size for the object. This shows the degree of packing performed by the system under test.

Tables 21, 22, and 23 list the ACEC Pragma PACK benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. Additional benchmarks are not needed for Pragma PACK.

4.7.2 Length Clause: SIZE Specification Benchmarks

Tables 24 and 25 list the ACEC Length Clause SIZE specification benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers. In these test cases, an integer type is declared and then the TYPE'SIZE representation clause is used to determine the size in bits an object of that type can occupy. An array is declared of that integer type. Tests are then performed on components of that array.

Comments:

1. Additional benchmarks are not needed for this feature.

4.7.3 Record Representation Clause Benchmarks

In these tests, the record representation clause is used to specify the layout of a record whose components are boolean variables (that have the SIZE representation clause specified) as well as a packed boolean array.

Table 26 lists the ACEC Record Representation Clause benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments:

1. Additional benchmarks are not needed for this feature.

4.7.4 Attribute Tests

These tests determine if certain attributes have been implemented. Table 27 lists the ACEC Attribute tests along with the results of running these benchmarks on the HP and Verdix Ada compilers. The attributes tested include:

1. Test ADDRESS attribute of a subroutine, local object, and dynamic object.
2. Test SIZE attribute of a local object, and dynamic object.
3. Test POSITION, FIRST_BIT, and LAST_BIT attribute for a record component.
4. Test STORAGE_TYPE attribute for an access type and task type.

Comments:

1. Additional benchmarks are not needed for this feature.

4.7.5 Unchecked_Conversion

In real-time systems, it is very frequently required to do a unchecked_conversion from one type to another. These set of benchmarks test the time required to do a unchecked_conversion from one type to another. Table 28 lists the unchecked_conversion benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: None.

Additional Unchecked_Conversion Benchmarks:

1. In many real-time systems a string object may be converted to an integer via unchecked_conversion as well as an array of floats may be converted to a record of floats.

   Benchmark: Measure the time for UNCHECKED_CONVERSION to move a STRING object to another INTEGER object.

2. *Benchmark: Measure the time to do an unchecked conversion of an array of 10 floating components into a record of 10 floating components.*

4.8 Interrupt Handling

In real-time embedded systems, efficient handling of interrupts is very important. Interrupts are critical to the ability of the system to respond to real-time events and perform its required functions and it is essential that the system responds to the interrupt in some fixed amount of time.

Table 29 lists the ACEC Interrupt handling benchmarks.

Comments: None.

Additional Interrupt Handling Benchmarks:

1. In many real-time systems, it is important that interrupts are not lost when an interrupt is being handled and another interrupt is received from the same device.

   Benchmark: Determine if an interrupt is lost when an interrupt is being handled and another interrupt is received from the same device.

2. An implementation may cause scheduling decisions on receipt of an interrupt. This may not be desirable in some real-time systems.

   Benchmark: Determine if an interrupt entry call invokes any scheduling decisions.

3. The handler deals with high priority interrupts, and is therefore allocated a high task priority. However, it can be interrupted outside the rendezvous by a low priority interrupt and cannot guarantee to return to the accept statement in time to catch the next high priority interrupt.

   Benchmark: Determine if accept statement executes at the priority of the hardware *interrupt, and if priority is reduced once a synchronization point is reached following the completion of accept statement.*

4.9 Pragmas

There are certain predefined pragmas which are expected to have an impact on the execution time and space of a program. These include: SUPPRESS, OPTIMIZE, SHARED, INLINE, PACK, CONTROLLED, and PRIORITY. Benchmarks for Pragma INLINE are covered under Subprogram Overhead in Section 4.10. Pragma PACK is covered under Chapter 13 benchmarks (Section 4.7), Pragma CONTROLLED is covered under Memory Management benchmarks (Section 4.2), and Pragma PRIORITY is covered under Tasking (Section 4.1).

4.9.1 Pragma SUPPRESS

The benchmarks for pragma SUPPRESS determine the improvement in execution time when pragma SUPPRESS is used. Pragma SUPPRESS causes the compiler to omit the corresponding exception checking (RANGE_CHECK, STORAGE_CHECK etc.) that occurs at runtime.

Table 30 lists the ACEC Pragma SUPPRESS benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: None.

Additional Pragma SUPPRESS Benchmarks:

1. Timing has also to be measured for other kinds of checks using Pragma SUPPRESS.

   Benchmark: Pragma SUPPRESS is used for these checks: Access_Check, Index_Check, Length_Check, Storage_Check, and Elaboration_Check.

4.9.2 Pragma OPTIMIZE

The benchmarks for pragma OPTIMIZE determine the improvement in execution time when the pragma is used.

Comments:

1. ACEC has no pragma OPTIMIZE benchmarks.

Additional Pragma OPTIMIZE Benchmarks:

1. Timing has to be measured for improvement in execution time pragma OPTIMIZE is used with options TIME and SPACE.

   Benchmark: Determine improvements in execution time when pragma OPTIMIZE is used with options TIME and SPACE.

4.9.3 Pragma SHARED Benchmarks

With multiple tasks executing, there may be an instance where the same nonlocal variable must be accessed. Pragma SHARED is the mechanism that designates that a variable is shared by two or more tasks. Pragma SHARED directs the RTE to perform updates of the shared variable copies each time they are updated, but the overhead may be significant.

Comments:

1. There are no ACEC Pragma SHARED benchmarks.

Additional Pragma SHARED Benchmarks:

1. The overhead involved in updating a shared integer variable is compared to the overhead involved in updating an integer variable that is not shared.

   Benchmark: Determine the overhead due to Pragma SHARED when two tasks access a shared integer variable.

   The main program updates a shared integer variable. This integer variable is also updated by another task.

2. The overhead involved in updating a shared integer variable during a rendezvous is compared to the overhead involved in updating an integer variable (that is not shared) during a rendezvous.

   Benchmark: Determine the overhead in rendezvous time when a shared variable is updated during the rendezvous.

4.10 Subprogram Overhead

In Ada, subprograms rank high among program units from a system structure point of view. If the subprogram overhead is high, then the compiler can generate INLINE expansion at the cost of increasing the size of the object code. However, if calls to that subprogram are made from a lot of places, then the pragma INLINE defeats the purpose due to increase in size of object code.

Tables 31 and 32 list the ACEC Subprogram overhead tests along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: None.

Additional Subprogram Overhead Tests:

1. Subprogram overhead timings have to be measured when various parameters are passed during a procedure call.

   Benchmark: Various numbers of parameters of types INTEGER and ENUMERATION are passed to determine the subprogram overhead associated with simple parameter passing. Then composite objects (arrays and records) are passed to determine if they are passed by copy or reference. Finally, the subprogram is called with formal parameters of an unconstrained composite type.

   All of the tests include passing the parameters with modes **in**, **out**, and **in out**. All of the tests involve two different types of subprogram calls, one to a subprogram that is a part of the same package as the caller, and the other to a subprogram in a package other than the one in which the caller resides.

2. Benchmarks for subprogram overhead that involve the use of package instantiations of generic code are also needed.

   Benchmark: All of the above tests for both inter-package and intra-package procedure calls are repeated with the subprograms being part of a generic unit.

4.11 Numeric Computation

An embedded system must be able to represent real-world entities and quantities to perform related manipulations and computations. There should be support for numerical computation, units of measure (including time), and calculations and formula from physics, chemistry etc. Numeric computation benchmarks are discussed under two separate logical headings: a) Mathematical Computation Benchmarks and b) Benchmarks for Arithmetic on Type TIME and DURATION. The ACEC suite has over 300 numeric computation benchmarks which belong to different categories. Hence, these benchmarks have not been presented in the form of tables.

4.11.1 Mathematical Computation Benchmarks

These benchmarks range from simple integer additions to calculation of complex mathematical equations. ACEC has the following categories of mathematical computation benchmarks:

1. Floating point addition, multiplication, division

2. Integer addition, division, multiplication
3. Natural Integer addition, division, multiplication, mod, rem
4. Fixed point addition, multiplication, division
5. Type conversions from integer to real, floating point literal to integer, integer to float, convert one fixed point to another, convert double to real, convert real to double
6. Integer addition, division
7. exp, ln, sin, cos, abs, sqrt, atan, sgn
8. mod, and rem operators
9. polynomial evaluation
10. Long integer assignment, addition, subtraction, multiplication, division, rem, conversion from integer to long integer
11. extended precision floating point assignment, addition, division, abs, sin, cos, exp, ln, sqrt, atan, extended precision floating point array assignment

Comments:

1. As far as mathematical computation benchmarks are concerned, the ACEC set is comprehensive and complete.

4.11.2 Benchmarks For Arithmetic On Type TIME and DURATION

For real-time embedded systems, it is necessary to dynamically compute values of type TIME and DURATION. An example of such a computation is the difference between a call to the CLOCK function and a calculated TIME value. This value may be used as a parameter in the delay statement. If the overhead involved in this computation is significant, the actual delay experienced will be longer than anticipated which could be critical for real-time systems.

ACEC benchmarks for arithmetic on type TIME and DURATION are divided in the following categories:

1. Addition of variables of type CALENDAR.TIME.
2. Comparison of variables of type CALENDAR.TIME.
3. Comparison of type DURATION with SECONDS(TIME).
4. Call on CALENDAR.TIME_OF function.

Comments: None.

Additional Benchmarks For Arithmetic On Type TIME and DURATION:

1. Many real-time systems may have the need to compute values using the "+" and "-" functions provided in the package calendar.

   Benchmark: Measure the overhead associated with a call to and return from the "+" and "-" functions provided in the package CALENDAR.

   Times are measured for computations involving just variables and both constants and variables of Type TIME and DURATION. The variables have predefined values. Although both "+" functions are essentially the same (only the order of parameters reversed) both are tested. This is done because a discrepancy in the time needed to complete the computation will occur if one of the functions is implemented as a call to the other.

4.12 Real-Time Paradigms

Users, system programmers, and academicians have found a number of useful paradigms for building concurrency. These real-time paradigms can be coded in Ada and benchmarked.

Table 33 lists the ACEC real-time paradigms along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: None.

Additional Real-time Paradigms:

1. Many real-time implementations require buffered and unsynchronized communication between tasks. Due to the rendezvous being a synchronous and unbuffered message passing operation, intermediary tasks are needed to uncouple the task interaction to allow tasks more independence and increase the amount of concurrency. Various combinations of intermediary tasks are used in different task paradigms to create varying degrees of asynchronism between a producer and consumer. The benchmarks defined here evaluate the cost of introducing intermediary tasks for various real-time tasking paradigms. The goal of these benchmarks is to give real-time programmers a feel for the cost of using such paradigms in a real-time embedded application and to avoid using such paradigms if the cost is unacceptable for a real-time system.

   Benchmark: Measure the cost of rendezvous between a producer and consumer. The task that is the source of the information is called the producer and the task that is the recipient of the information is called the consumer.

   Benchmark: Measure the cost of rendezvous using buffer tasks. A buffer is pure server task that provides for one entry for storing of items in a buffer and another entry for providing items from the buffer. Both the consumer and the producer call the buffer task to obtain a piece of information.

Benchmark: Measure the cost of rendezvous using a buffer and and transporter. This scheme uses two intermediary tasks between the producer and the consumer.

Benchmark: Measure the cost of rendezvous using a buffer and and two transporters. If both the producer and consumer wish to communicate via a buffer and both need to be called tasks, it is necessary to use a transporter on each side of the buffer. This results in the producer-transporter-buffer-transporter-consumer paradigm.

2. A monitor is commonly used for controlling a system's resources. For example, read and write operations to a disk are usually controlled by a monitor that ensures the integrity of data on the disk. This is also known as mutual exclusion. Monitors can be implemented to have controlled access to a shared data pool. Monitors can be implemented via semaphores, event signaling, and rendezvous mechanism. The implementation via semaphores and event signaling is essentially the same.

   Benchmark: Measure time to access data in a pool using a monitor.

   Any number of processes are allowed to read the pool simultaneously, but no reads are permitted during a write operation. The monitor developed is used to control the reading and writing of data to the pool. Two implementations of the monitor can be considered: the first using semaphores, and the second using the Ada rendezvous mechanism.

4.13 Composite Benchmarks

A composite benchmark is defined as a program, within the context of the application domain, that looks at the interaction between Ada features rather than the performance of individual features themselves. The purpose of running a composite benchmark is to aid in the selection of a suitable compiler and runtime for a particular application.

The ACEC also includes examples from actual application code. This code contains test problems representative of how Ada is being used in practice. These are example test problems drawn from Ada programs extracted from projects. Table 34 lists those benchmarks along with the results of running these benchmarks on the HP and Verdix Ada compilers.

Comments: Observations about the ACEC composite benchmarks are:

1. The code contained in the composite benchmarks is not very useful and does not provide any relevant information for compiler evaluation for real-time systems.

Additional Composite Benchmarks:

1. Composite benchmarks can be developed for a number of individual applications such as Intelligence/Electronic Warfare (IEW) systems, avionics systems, etc. More information about the process of development of composite benchmarks is contained in a report titled "Ada Composite Benchmark for Intelligence/Electronic Warfare Systems" [11].

   Benchmark: Develop composite benchmarks for IEW systems [11].

4.14 Portability Of the ACEC Benchmarks

The ACEC tests essentially consist of two sets of tests:

a. Tests that do not depend on the MATH packages supplied with the ACEC (e.g. the tasking tests)
b. Tests that depend on the MATH packages.

The tests that do not depend upon the MATH packages can easily be ported from one Ada compiler to another. More effort is required to port tests that depend upon the MATH packages. This section describes the steps that have to be performed to port the ACEC to any Ada compiler configuration. The ACEC is intended to run on bare targets as well as targets with operating systems.

4.14.1 Modification To the Command Files

The first step is to modify the command files that run the ACEC benchmarks. The command files cmp.unx, cmp_tst.unx and cmp_base.unx that run the ACEC benchmarks have to be modified to reflect the compilation and linking commands of the Ada compiler system.

Porting Effort: Easily portable.

4.14.2 Modification To the Base ACEC Files

The second step involves modification to the a set of files known as the base ACEC files. These are global.ada, math_dep.ada, math_test.ada, and dbl_mathtest.ada.

1. *global.ada:*

   An ACEC user can choose to run the timing loop using CPU time rather than elapsed time. Using CPU time permits the collection of measurements on multiprogramming target systems without having to shut the system down to eliminate contending jobs. The ACEC benchmarks utilize the function CPU_TIME_CLOCK which is defined in the package GLOBAL in the file global.ada. This function is implementation dependent and has to be written for each Ada compiler system.

If the system does not support an integer type with at least 32 bits of precision, the declaration of the type "BIGINT" and "BIGNAT" will not compile and must be removed.

If the system does not support a floating point type with 9 digits of precision, the declaration of the type "DOUBLE" will not compile and must be removed.

The function "ADDRESS_TO_INT" converts a value of type SYSTEM.ADDRESS to an integer type. This function is used to compute the code expansion sizes by subtracting the address values of two label'ADDRESS attributes (or of two type ADDRESS variables obtained by a GETADR function). On different systems, SYSTEM.ADDRESS'SIZE will differ forcing a modification to the return type of this function.

Porting Effort: Easily implementable.

2. *math_dep.ada:*

   The package MATH_DEPENDENT in math_dep.ada has to be adapted to reflect both the characteristics of the target machine floating point hardware and the facilities which the Ada compilation system provides to manipulate bit fields in floating point variables. The size and location of the sign, exponent, and mantissa of a floating point number are critical, as are other representation details such as the encoding of the exponent field.

   Porting Effort: Non-trivial.

3. *math_test.ada and dbl_mathtest.ada:*

   The file mathtest.ada has the program MATHTEST which tests the math routines. MATHTEST requires a package MACHINE which contains some hardware dependent constants which are modified for the ACEC benchmarks. These values are obtained from the compiler documentation.

   The file dbl_mathtest.ada has the program DBL_MATHTEST which tests the double precision math routines. DBL_MATHTEST requires a package DBL_MACHINE which contains some hardware dependent constants which are modified for the ACEC benchmarks. This information can be obtained from the documentation for the HP Ada compiler system.

   Porting Effort: Non-trivial.

4.14.3 Input/Output

The third step is to make sure that TEXT_IO and FLOAT_IO are supported on the target on which the ACEC suite has to be run. The ACEC outputs strings containing numeric results of performance tests. If TEXT_IO and FLOAT_IO are not supported, the results of the tests cannot be displayed without modification to the timing code loop. If FLOAT_IO is not directly supported, users will have to develop work arounds.

If the Ada system supports a complete version of the TEXT_IO package, this requirement causes no problems. However, some Ada compilers targeted to real-time systems (where the hardware is limited) may limit their I/O facilities. The standard timing output is in microseconds to the nearest tenth. What is needed is the capability to output real numbers. If thirty-two bit integers are available, one viable option is to multiply timing results by 10 and then output INTEGR'IMAGE. A user may have to write a floating point to a text-string conversion routine.

For bare machine implementations of Ada, the effort required to get TEXT_IO to work well enough to output results of the timing and sizing measurement on a console can be large. Portions of the Ada runtime library may need to be modified, I/O device drivers may need to be written and tested.

Porting Effort: Non-trivial.

5. Conclusions

Benchmarking Ada implementations to determine their suitability for real-time systems is an extremely complex task. This job is made even more difficult due to differing requirements of various real-time applications. The ACEC benchmarks provide a good start for benchmarking Ada compilers meant for real-time applications. However, the ACEC benchmarks need to be augmented with more benchmarks in certain areas as outlined in this report. It is hoped that the results of this study will enable appropriate extensions to the ACEC benchmark suite so that they are more useful in benchmarking Ada compilers meant for real-time systems.

REFERENCES

[1] Ada Compiler Evaluation Capability (ACEC) Technical Operating Report (TOR) User's Guide, Report D500-11790-2, Boeing Military Aerospaces, P.O. Box 7730, Wichita, Kansas, 1988.

[2] Ada Compiler Evaluation Capability (ACEC) Version Description Document, Report D500-11790-3, Boeing Military Airplane, P.O. Box 7730, Wichita, Kansas, 1988.

[3] CECOM Center for Software Engineering, "Establish and Evaluate Ada Runtime Features of Interest for Real-time Systems", C02092LA0003, Final Report delivered by IITRI, 15 Feb 1989.

[4] CECOM Center For Software Engineering, "Real-time Performance Benchmarks For Ada", C02-092LY-0001-11, Final Report delivered by Arvind Goel (TAMSCO), 24 March, 1989.

[5] R.M. Clapp et al., "Towards Real-time Performance Benchmarks for Ada", CACM, Vol. 29, No. 8, August 1986.

[6] N. Altman, "Factors Causing Unexpected Variations in Ada Benchmarks", Software Engineering Institute Technical Report, CMU/SEI-87-TR-22, October 1987.

[7] N. Altman et al., "Timing Variation in Dual Loop Benchmarks" , Software Engineering Institute Technical Report, CMU/SEI-87-TR-21, October 1987.

[8] CECOM Center for Software Engineering, "Catalogue of Ada Runtime Implementation Dependencies", C02 092JB 0001, Final Report delivered by LabTek, (revised ARTEWG Document), 15 Feb 1989.

[9] "Catalogue of Interface Features and Options for the Ada Run Time Environment", ARTEWG Report, 1989.

[10] CECOM Center For Software Engineering, "Performance Measurements of the CHS Ada Compiler", Final report delivered by Unixpros Inc., 15 December, 1989.

[11] CECOM Center For Software Engineering, "Ada Composite Benchmark for Intelligence/Electronic Warfare Systems", Draft Final Report delivered by Unixpros Inc., July 10, 1990.

Appendix A: ACEC Real-time Benchmarks and Execution Results

Appendix A lists the ACEC real-time benchmarks. It also presents the results of running the ACEC real-time benchmarks on the following Ada compilers:

- HP-Ada Compiler (Releases 3.25 and 4.35) running on HP 9000/350 machine under HP-UX Release 6.2.
- Verdix Ada Compiler (Release 5.41) hosted on a Sun 3/60 and targeted to a Motorola 68020 bare machine.

The hardware and software configurations for the two compilers is as follows:

HP Testbed Hardware and Software

The hardware used for benchmarking was Hewlett-Packard 9000/350 CPU running HP-UX V 6.2. The set up can be summarized as follows:

Host: HP 9000/350 running HP-UX V 6.2.

Compiler: Self-hosted HP (basically the Alsys Ada Compiler) Ada Development System Version 3.25 and 4.35.

Target: Same as the host.

Verdix Testbed Hardware and Software

The hardware used for benchmarking was Sun 3/60 CPU running Sun Unix 4.2 Release 3.5, linked to a single 12.5 Mhz Motorola 68020 single board computer enclosed in a multibus chasis. The setup can be summarized as follows:

Host: Sun 3/60, running Sun Unix 4.2 Release 3.5

Compiler: Verdix Ada Development System targeted to Motorola MC68020 targets, release 5.41

Target: GPC68020 (based on Motorola MC68020 microprocessor) multibus-compatible computer board having 12.5 Mhz MC68020 microprocessor, a MC68881 floating point co-processor, and 2 megabyte of RAM.

The interrupt handling benchmarks were not executed due to lack of the required hardware to produce external interrupts. A comprehensive performance evaluation of the HP Ada compilers obtained by running the ACEC and Real-time benchmarks is presented in reference [10]. For the Verdix compiler, comparing the ACEC real-time benchmark results to the Real-time benchmarks (listed in [4]) shows that the two sets of results are in sync for the common features measured by those benchmarks.

TABLE 2. Task Activation/Termination

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task1	Task creation and termination without the new operator	14281.5	10079.2	4800.0
task2	Measure task activation/termination timing where 10 task objects of a single type are declared in a procedure that is called from the main program	10908.08	7052.74	4800.0
task51	Measure task activation/termination timing where a task object is declared as part of a record which is then created via the new allocator. Timing includes both allocation/deallocation timings for a task.	14221.7	11682.2	4500.0

TABLE 3. Measure Time For Simple Rendezvous

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task3	Task priorities are such that task making entry call arrives at the rendezvous first	3484.5	3213.3	420.7
task23	Task priorities are such that task doing accept arrives at the rendezvous first	3482.4	3230.2	357.7
task24	Task priorities are such that task making entry call arrives at the rendezvous first, task performing accept is in a subunit	3474.2	3189.2	420.7
task26	Task priorities are such that task making the entry call arrives at the rendezvous first, task performing accept is in a separate package	3454.6	3227.5	419.9
task41	Task priorities are such that task doing accept arrives at the rendezvous first, task performing the accept is in a separate package	3475.1	3270.8	357.7
task42	Task rendezvous between equal priority tasks, task performing accept in a separate package	1744.4	1736.7	354.6
task43	Task rendezvous between equal priority tasks, both tasks in same compilation unit	1765.1	1715.2	354.6
task49	Task entry call to one of an entry family, task priorities such that task making entry call arrives at rendezvous first	3164.9	3429.7	423.5
task52	Task rendezvous with a task created via the new allocator, task priorities are such that task making entry call arrives at rendezvous first. Time of task craetion/termination is excluded.	3399.1	3202.6	420.7
task57	Task rendezvous with task that is passed as a parameter to a subprogram, task priorities are such that task making entry call will arrive at rendezvous first	3443.9	3241.9	426.1

TABLE 4. Rendezvous Performance With Varying Number of Tasks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task_num_30	Rendezvous performance with Varying Tasks, 60 entry calls 30 different tasks	2603.0	2437.1	375.1
task_num_25	Rendezvous performance with Varying Tasks, 50 entry calls 25 different tasks	2615.0	2452.0	367.0
task_num_20	Rendezvous performance with Varying Tasks, 40 entry calls 20 different tasks	2606.4	2440.1	374.1
task_num_15	Rendezvous performance with Varying Tasks, 30 entry calls 15 different tasks	2621.3	2417.4	367.1
task_num_10	Rendezvous performance with Varying Tasks, 20 entry calls 10 different tasks	2628.0	2444.7	369.2
task_num_5	Rendezvous performance with Varying Tasks, 10 entry calls 5 different tasks	2612.0	2402.1	376.1
task_num_1	Rendezvous performance with Varying Tasks, 2 entry calls 1 different tasks	3425.8	3214.8	357.1
task2_num_30	Rendezvous performance with Varying Tasks, 30 different tasks, 31 entry calls are queued up on one accept before being processed	3504.1	3262.3	420.1
task2_num_25	Rendezvous performance with Varying Tasks, 25 different tasks, 26 entry calls are queued up before being processed	3487.0	3286.3	419.1
task2_num_20	Rendezvous performance with Varying Tasks, 20 different tasks, 21 entry calls are queued up before being processed	3452.0	3247.5	416.3
task2_num_15	Rendezvous performance with Varying Tasks, 15 different tasks, 16 entry calls are queued up before being processed	3512.7	3214.8	418.5
task2_num_10	Rendezvous performance with Varying Tasks, 10 different tasks, 11 entry calls are queued up before being processed	3541.5	3237.5	419.3
task2_num_5	Rendezvous performance with Varying Tasks, 5 different tasks, 6 entry calls are queued up before being processed	3497.6	3240.9	415.1
task2_num_1	Rendezvous performance with Varying Tasks, 1 different task, 2 entry calls are queued up before being processed	3481.0	3218.0	410.1

TABLE 5. Select Statement with Else Alternative

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task5	Select statement makes an entry call that never succeeds, ELSE alternative always executed	100.4	73.5	39.0
task32	Select statement has an accept statement that is never called, ELSE alternative always executed	661.8	436.4	39.0
task33	Select statement has an accept statement that is never called, ELSE alternative always executed	667.6	443.0	39.0

TABLE 6. Rendezvous Calls with Conditional Selects

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task15	Accepting task contains a conditional select with no open alternatives, accepting task arrives at rendezvous first	765.7	545.8	81.5
task16	Accepting task contains a conditional select statement with no open alternative, no waiting tasks, selected alternative is DELAY -1	848.1	592.6	263.2
task17	Accepting task contains a conditional select statement with all open alternative, no waiting tasks, selected alternative is ELSE	1040.2	835.5	83.4
task18	Accepting task contains a conditional select statement with all closed alternatives, waiting tasks, selected alternative is ELSE	664.4	522.5	380.7
task19	Accepting task contains a conditional select statement with one satisfied alternative, waiting tasks on all entries	20291.4	19577.2	3219.7
task21	Entering task makes a sequence of entry calls, accepting task contains conditional select with guards, only one of which is satisfied, entering task arrives at rendezvous first	3975.5	3982.2	555.9
task22	Entering task makes a sequence of entry calls, accepting task contains conditional select with guards, only one of which is satisfied, accepting task arrives at rendezvous first	4565.0	4088.2	544.3

TABLE 7. Selective Wait

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task30	Selective wait in accepting task, task waiting on accept and DELAY alternative will not be taken	5470.9	4895.0	1979.5
task31	Selective wait in accepting task, DELAY alternative taken and then canceled to make rendezvous	7699.5	7178.7	2102.1
task34	Selective wait in accepting task, always takes the DELAY 0.001 alternative which expires without entry call being made	Error	Error	1344.3
task34	Time for DELAY 0.001 seconds	Error	Error	1344.3
task35	Selective wait in accepting task, always takes the DELAY 0.0 alternative which expires without entry call being made	696.2	431.3	289.6
task35	Time for delay 0.0	56.6	43.0	169.2
task59	Selective wait in accepting task, multiple delay -1 alternatives one of which is executed	1031.7	848.0	1130.8

TABLE 8. Exceptions Raised During Rendezvous

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task37a	User-defined exception is raised inside a rendezvous, time measured is the time to create two tasks, have a rendezvous, raise and handle the user-defined exceptions in both the tasks and then terminating the tasks	56616.8	46439.2	Error
task37b	Difference between task37a and task37b is the incremental time in task37a to raise and propagate an exception from within a rendezvous	31227.7	28105.6	Error
task38	Entry call made to a task that was aborted, TASKING_ERROR is raised and then handled inside the block statement, time to raise TASKING_ERROR and handling the exception inside the block statement	8331.0	5308.8	Error
task54	Task specifies an inadequate storage_size with a static expression, resulting exception is handled with a null statement	Error	Error	Error
task55	Task specifies an inadequate storage_size with a dynamic expression, resulting exception is handled with a null statement	Error	Error	Error

Note: The error raised in some benchmarks in this table is due to the large number of iterations (thus requiring more memory space for tasks) required by the ACEC benchmarks and thus running out ~f memory space.

TABLE 9. Abort Statement

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task39	Main program aborts a task that is already aborted	150.7	104.7	87.6
task40	Program creates a task which aborts itself	22772.3	15154.6	Error
task53	One task aborts another task, aborted task created on each iteration	25980.0	20150.4	Error

Note: The error raised in some benchmarks in this table is due to the large number of iterations (thus requiring more memory space for tasks) required by the ACEC benchmarks and thus running out of memory space.

TABLE 10. Miscellaneous Tasking Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task58	Determine if accepts are done in lexical order	No	No	Yes
task60	Determine if accepts are done in lexical order	No	No	Yes
task6	Time it takes to determine task attributes CALLABLE or TERMINATED	39.6	32.6	22.1

TABLE 11. Habermann-Nassi Tasking Optimizations

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task11	One task calls entries in another task one after another. Tasks priority is such that the accepting task will arrive at the rendezvous first. Habermann-Nassi optimization can be applied as there are no statements to be executed outside the accept statements	3740.6	3311.9	363.6
task12	One task calls entries in another task one after another. The task accepting the entry calls has the accept statements in a select and it arrives at the rendezvous first. Habermann-Nassi optimization can be applied as there are no statements to be executed outside the accept statements	4745.8	4299.3	493.7
task13	One task calls entries in another task one after another. The task accepting the entry calls has the accept statements in a select and it arrives at the rendezvous first. There is some code outside of the rendezvous and this requires a more complex treatment (to Habermann-Nassi-ize) that forces the server task to maintain an independent thread of control	3813.6	3323.5	375.6
task14	One task calls entries in another task one after the another. The task accepting the entry calls has the accept statements in a select and it arrives at the rendezvous first. It also has WHEN clause on the accepts only one of which is satisfied. Habermann-Nassi optimization can be applied with slightly more difficulty	3946.5	4000.3	552.4
task20	Entering tasks makes sequence of entry calls. Accepting task contains a SELECT statement with WHEN clauses, only one of which is satisfied. Entering task will arrive at rendezvous first. Habermann-Nassi optimization can be applied with slightly more difficulty	4576.2	4098.3	542.2

Note: Habermann-Nassi optimizations is not implemented for the HP and Verdix Compilers.

TABLE 12. Other Tasking Optimizations

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task27	Redundant Code Elimination - Simple Rendezvous with the task making an entry call arriving at rendezvous first. Brackets each ACCEPT with a SELECT/END pair. Redundant code elimination Select with one unconditional accept can be treated as a simple accept	3711.7	3823.3	448.7
task28	Simple Rendezvous with the task making an ENTRY call arriving at the rendezvous first. Brackets each ACCEPT with a SELECT/ END pair containing WHEN clauses which can be evaluated at compile time. Can be folded and simplified	3763.9	3816.9	470.1
task29	Folding of WHEN Clause Conditions - Simple Rendezvous with the task making an entry call arriving at rendezvous first. Brackets each ACCEPT with a SELECT/END pair containing WHEN clauses which can be evaluated at compile time	4358.3	3837.3	618.5

TABLE 13. Memory Allocation Timing Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss22	Time to allocate a statically bounded one dimensional array (size 500) of float, and assign to component	5.5	6.2	12.9
ss23	Time to allocate a statically bounded two dimensional array (10 X 10) of float, and assign to component	8.4	10.5	18.4
ss24	Time to allocate a statically bounded three dimensional array (5 X 5 X 5) of float, and assign to component	18.6	20.2	37.9
ss25	Time to allocate a dynamically bounded one dimensional array (size 2) of float, and assign to component	12.4	14.7	35.2

TABLE 14. Memory Allocation/Deallocation Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss162	Time to allocate 100 linked entries from heap, follow chain then deallocate	18020.6	9531.9	7919.7
ss163	Allocate 100 linked entries from heap, immediately deallocate, an optimizing compiler can omit deallocation	13877.7	7738.2	7053.0
ss164	Allocate 100 link objects in collection and immediately deallocate them, 100% padding allowed in collection	7955.8	3991.1	STORAGE_ERROR
ss165	Allocate 100 link objects in collection and follow chain to explicitly deallocate them 100% padding allowed in collection	7568.1	4240.1	STORAGE_ERROR
ss166	Allocate 100 linked objects in collection and follow links, does not deallocate, relying on whole collection being freed when block is exited. Time difference between ss166 and ss165 is time to deallocate 100 objects	6260.9	3631.6	STORAGE_ERROR
ss167	Allocate 100 linked objects in collection. Does not deallocate, relying on whole collection being freed when block is exited. Specifies pragma(CONTROLLED)	5670.1	3302.1	STORAGE_ERROR
reclaim	Check for reuse of reclaimed space when an ACCESS type to a constrained object type is allocated and then deallocated in global heap. Determine whether space is always, never, or sometimes immediately reused	Always	Always	Always
reclaim	Check for reuse of reclaimed space when an ACCESS type to a unconstrained object type is allocated and then deallocated in global heap. Determine whether space is always, never, or sometimes immediately reused	Always	Always	Always
reclaim	Check for reuse of reclaimed space when an ACCESS type to a constrained object type is allocated and then deallocated in a collection. Determine whether space is always, never, or sometimes immediately reused	Always	Always	STORAGE_ERROR
reclaim	Check for reuse of reclaimed space when an ACCESS type to a constrained object type is allocated and then deallocated in a collection. Determine whether space is always, never, or sometimes immediately reused	Always	Always	STORAGE_ERROR

TABLE 15. Exception Handling

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss153	Time to raise CONSTRAINT_ERROR implicitly and handling the exception in a block statement	2134.1	1423.2	5643.0
ss602	Time to raise CONSTRAINT_ERROR implicitly and handling the exception in a block statement	2503.8	1755.9	5876.0
ss369	Time to raise NUMERIC_ERROR implicitly (divide by zero) and handling the exception in a block statement	2077.9	1489.8	5655.0
ss311	Time to declare user-defined exception, raise it and then handle it in a block statement	2118.8	1392.5	2966.0
ss312	User-defined exception is just declared in a block statement, it is not raised	1.0	0.8	1.9
ss313	User-defined exception neither declared nor raised	2.4	0.8	2.0
ss379	Procedure h1 calls another procedure g1 which calls another procedure f that raises user-defined exception, exception is propagated to g1 and then raised again in g1 which is propagated and handled in h1	7820.2	5249.8	11688.0
ss380	Procedure h2 calls another procedure g2 which calls another procedure f that raises user-defined exception, exception is propagated to g2 and then propagated to h2 and then handled in h2	5161.8	3311.7	8305.0
ss381	Block with exception handler which calls on a procedure which raises the exception, the procedure it calls on does not have a handler but simply raises the exception	3381.8	2307.4	5628.0
ss382	Make two procedure calls. The lowest level has an exception handler which can (re) raise an exception and propagate it to the next higher level. This problem does not raise the exception. In this problem the exception is NOT raised.	10.8	12.0	27.1
ss383	Make two procedure calls. The lowest level does not have an exception handler and will simply propagate exception raised to the next higher level. This problem does not raise the exception.	10.5	10.2	24.9
ss384	Call on procedure which does'nt propagate exception	5.6	6.2	12.8
ss528	Conditionally raise user-defined exception and go through handler	2371.5	1534.9	4628
ss527	Same as ss528, except that the exception is not raised	4.7	4.6	12.0
funcexcp	Purpose of test is to measures the time associated with cleaning up the stack when an exception is raised during a nested function call. A poor implementation may leave garbage on the stack associated with function call and actual parameters.	4763.0	3302.6	6825.0

TABLE 16. TEXT_IO Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
io0	Time to do Set_Col on named file	555.9	240.7	71.3
io1	Time to open/close a text file	198302.5	223188.3	845937.5
io2	Time to open file, put 1000 80-character lines, and close file	2162862.5	2621403.2	285047008.0
io3	Time to open file, get 1000 80-character lines written in io2, and close file	45219200.0	47024732.0	68143976.0
io4	Time to open file, use get_line to read the 1000 80-character records written in io2, and close file	35566752.0	39014428.0	67994008.0
io5	Time to open file, use put_line to write 1000 80-character records and close file	1724241.0	1960474.0	284981024.0
io6	Time to open file, use put to write 100 512-bytes records, close the file The records contain 1 6-byte count field, and 406 bytes of blanks	396501.1	526569.9	86130976.0
io7	Time to open file, use get_line to read the 100 512-byte records written in io6	22545352.0	24655966.0	86130976.0
io8	Time to access the end-of_file function	26.7	22.2	58.5
io9	Time for the reset function	66.4	61.9	472036.1.0
io10	Time for is_open function	16.6	4.5	8.9

TABLE 17. DIRECT_IO Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
io11	Time for set_index function	24.5	15.2	74310.4
io12	Set_Index function followed by a READ. Read from same block so no physical IO operations are required	647.6	643.5	290029.0
io13	Set_Index function followed by a WRITE. This will write to the same block each time	707.9	835.4	408005.3
io14	DIRECT_IO file OPEN/CLOSE	202828.5	224267.8	559316.1
io15	Time for the INDEX function	26.8	15.8	17.3
io16	Time for call on the SIZE function	377.7	305.0	200327.6

TABLE 18. SEQUENTIAL_IO Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
io17	Time to do OPEN/CLOSE on named file	181734.5	219260.0	564320.0
io18	Time to open file, write 1000 80 byte records and close file	1381535.0	1299983.0	283890976.0
io19	Time to open file, read 1000 80 byte records and close file, this reads file written in io19	1091817.0	1209198.2	529695968.0
io20	Time for end_of_file function	376.4	415.0	195647.3
io21	Time to open file, write 100 511 byte records, and close file	355088.9	480923.4	85826040.0
io22	Time to open file, read 100 80 byte records, and close file	323441.2	325373.9	85382984.0
io23	Time for set_input function	25.0	21.5	22.3

TABLE 19. CLOCK Function

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss452	Measure time for CLOCK function	1134.4	1220.3	927.6
ss453	Measure time for SECONDS function	1139.4	1225.5	1488.1

TABLE 20. Scheduling and Delay Statement

Time in Microseconds

Name	Description	HP	Verdix
task44	Timing of an entry call by a higher priority task which becomes eligible to run while two lower priority tasks are rendezvousing	10185.8	10185.2
task45	Timing of an entry call by a higher priority task which becomes eligible to run while a lower priority task is running	10203.1	10187.5
ss455	Time to execute delay 0.0	7.6	5.6
ss458	Time to execute delay 0.001	Unreliable	Unreliable
ss459	Time to execute negative delay statement	167.2	135.7
delay1	Delay 0.0 statement in higher priority task, contending tasks	52.1	45.9
delay8	Delay 0.0 statement in higher priority task, NO contending tasks	52.1	45.5
delay2	Delay 0.000001 statement in higher priority task, contending tasks	51.8	45.3
delay9	Delay 0.000001 statement in higher priority task, No contending tasks	51.8	45.5
delay3	Delay 0.000010 statement in higher priority task, contending tasks	51.8	51.9
delay10	Delay 0.000010 statement in higher priority task, No contending tasks	51.8	45.8
delay4	Delay 0.000100 statement in higher priority task, contending tasks	20401.3	20160.5
delay11	Delay 0.00010 statement in higher priority task, No contending tasks	20401.3	20160.5
delay5	Delay 0.00100 statement in higher priority task, contending tasks	20084.5	20144.0
delay12	Delay 0.0010 statement in higher priority task, No contending tasks	20084.5	20144.0
delay6	Delay 0.0100 statement in higher priority task, contending tasks	20402.2	20183.4
delay13	Delay 0.010 statement in higher priority task, No contending tasks	20402.2	20183.4
delay7	Delay 0.10 statement in higher priority task, contending tasks	117679.6	120684.8
delay14	Delay 0.10 statement in higher priority task, No contending tasks	117679.0	120684.8

TABLE 21. Pragma PACK

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss156	Field assignment to unpacked record	4.3	4.7	13.2
ss157	Field assignment to packed record	4.3	4.7	17.2
ss158	Time to assign to components of a unpacked record and then assigning that record to a packed record variable	8.2	8.6	16.3
ss159	Time to assign to components of a packed record and then assigning that record to a unpacked record variable	10.3	7.7	13.2
ss160	Time to assign to components of a unpacked record and then assigning that record to a unpacked record variable	7.2	7.0	17.5
ss161	Time to assign to components of a packed record and then assigning that record to a packed record variable	8.6	8.3	19.5
ss326	Time to perform =, AND and NOT operations on small unpacked boolean array	15.4	15.8	133.5
ss337	Time to perform =, AND and NOT operations on small packed boolean array	14.0	4.6	20.8
ss327	Time to perform =, and AND operations on small unpacked boolean array	11.8	12.0	77.6
ss338	Time to perform =, and AND operations on small packed boolean array	11.0	3.7	16.8
ss328	Time to perform /=, and AND operations on small unpacked boolean array	14.1	13.7	97.1
ss339	Time to perform /=, and AND operations on small packed boolean array	12.2	3.4	18.5
ss329	Time to perform AND operations on small unpacked boolean array	13.3	13.3	81.8
ss340	Time to perform AND operation on small packed boolean array	12.4	2.3	26.0
ss330	Time to perform OR operation on small unpacked boolean array	13.9	13.3	88.2
ss341	Time to perform OR operation on small packed boolean array	11.7	2.7	26.0
ss331	Time to perform OR operation on small unpacked boolean array, test uses a aggregate with range clause	43.2	40.1	171.4
ss342	Time to perform OR operation on small packed boolean array, test uses a aggregrate with range clause	40.1	3.4	224.0
ss332	Time to perform XOR operation on small unpacked boolean array	12.0	13.8	88.2
ss343	Time to perform XOR operation on small packed boolean array	11.8	2.3	26.0
ss333	Fetch from and store into array element operation on small unpacked array	2.2	2.2	3.9
ss344	Fetch from and store into array element operation on small packed array	2.2	3.1	10.6
ss334	Slice assignment operation on small unpacked array	4.9	4.6	20.9
ss345	Slice assignment operation on small packed array	4.5	11.9	9.3
ss335	Time to convert from packed to unpacked small boolean array	2.8	48.2	148.7
ss346	Time to convert from unpacked to packed small boolean array	3.9	41.6	188.5
ss336	Time to fetch element from small unpacked boolean array	1.7	2.2	2.7
ss347	Time to fetch element from small packed boolean array	1.9	1.7	5.6

TABLE 22. Pragma PACK (Continued)

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss348	Time to perform =, AND and NOT operations on large unpacked boolean array	191.5	191.1	624.3
ss351	Time to perform =, AND and NOT operations on large packed boolean array	191.6	70.6	88.8
ss349	Time to perform NOT, XOR, OR and AND operations on large unpacked boolean array	568.0	566.0	2336.9
ss352	Time to perform NOT, XOR, OR and AND operations on large packed boolean array	567.7	161.5	304.0
ss350	Element Assignment from packed to unpacked large boolean array	2.0	2.1	6.9
ss353	Element assignment from unpacked to packed large boolean array	1.8	2.3	10.3
ss652	Time for accessing a packed component spanning a storage unit boundary, bit size 3	14.1	13.8	22.1
ss657	Time for accessing a packed component spanning a storage unit boundary, bit size 5	13.4	13.9	error
ss662	Time for accessing a packed component spanning a storage unit boundary, bit size 7	13.5	13.5	Error
ss667	Time for accessing a packed component spanning a storage unit boundary, bit size 11	14.3	13.4	15.4
ss672	Time for accessing a packed component spanning a storage unit boundary, bit size 15	13.9	13.7	15.4
ss677	Time for accessing a packed component spanning a storage unit boundary, bit size 16	13.5	8.8	15.3
ss653	Time for accessing a packed component not spanning a storage unit boundary, bit size 3	13.5	13.5	22.1
ss658	Time for accessing a packed component not spanning a storage unit boundary, bit size 5	14.0	13.9	error
ss663	Time for accessing a packed component not spanning a storage unit boundary, bit size 7	13.5	13.5	Error
ss668	Time for accessing a packed component not spanning a storage unit boundary, bit size 11	13.5	13.8	14.9
ss673	Time for accessing a packed component not spanning a storage unit boundary, bit size 15	14.7	13.4	14.9
ss678	Time for accessing a packed component not spanning a storage unit boundary, bit size 16	13.9	8.7	14.9
ss654	Time for both spanning and nonspanning accesses, bit size 3	489.7	503.5	881.7
ss659	Time for both spanning and nonspanning accesses, bit size 5	482.5	505.0	Error
ss664	Time for both spanning and nonspanning accesses, bit size 7	524.2	512.8	Error
ss669	Time for both spanning and nonspanning accesses, bit size 11	517.5	505.3	617.7
ss674	Time for both spanning and nonspanning accesses, bit size 15	501.0	503.8	617.7
ss679	Time for both spanning and nonspanning accesses, bit size 16	508.1	337.7	617.7
ss655	Time for allocating a unpacked array component to a packed array component, bit size 3	116.2	226.9	Error

TABLE 23. Pragma PACK (Continued)

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss660	Time for allocating a unpacked array component to a packed array component, bit size 5	115.8	227.3	Error
ss665	Time for allocating a unpacked array component to a packed array component, bit size 7	116.7	228.4	Error
ss670	Time for allocating a unpacked array component to a packed array component, bit size 11	119.2	217.2	301.4
ss675	Time for allocating a unpacked array component to a packed array component, bit size 15	118.7	219.5	301.9
ss680	Time for allocating a unpacked array component to a packed array component, bit size 16	118.7	120.0	301.9
ss656	Time for simple assignment to a packed array component, bit size 3	12.2	10.0	Error
ss661	Time for simple assignment to a packed array component, bit size 5	12.4	10.0	Error
ss666	Time for simple assignment to a packed array component, bit size 7	12.2	10.0	Error
ss671	Time for simple assignment to a packed array component, bit size 11	12.2	10.2	10.8
ss676	Time for simple assignment to a packed array component, bit size 15	13.2	10.0	10.7
ss681	Time for simple assignment to a packed array component, bit size 16	12.2	7.4	10.8
ss682	Time to assign the left justified field of a packed boolean record to the NOT of its value	1.0	0.5	3.31
ss683	Time to assign the next to left justified field of a packed boolean record to the NOT of its value	0.9	1.0	4.4
ss684	Time to assign the next to right justified field of a packed boolean record to the NOT of its value	0.9	0.5	3.9
ss685	Time to assign the next to right justified field of a packed boolean record to the NOT of its value	1.0	0.5	4.4
ss686y	Time for packed array bit manipulation using array wide logical operators	2075.5	216.4	7121.9
ss686x	Time for packed array bit manipulation using array indexing	1572.1	226.5	5635.6
ss764	Time for bit manipulation using array aggregate. Set a component of packed boolean array to TRUE by using an OR against a variable	11.1	2.4	28.5
ss766	Time for bit manipulation using indexing. Set element of packed boolean array to true, selected by literal subscript	0.4	1.7	2.7
ss767	Time for bit manipulation using indexing. Set dynamically element of packed boolean array to true, selected by literal subscript	1.4	2.3	8.0
ss768	Time for bit manipulation using indexing. Set dynamically element of packed boolean array to true, using array of bits and OR operator	20.8	4.1	36.3

TABLE 24. SIZE Representation Clause Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss687	Time for accessing a packed component spanning a storage unit boundary, bit size 3	Error	13.8	Error
ss692	Time for accessing a packed component spanning a storage unit boundary, bit size 5	Error	13.8	Error
ss697	Time for accessing a packed component spanning a storage unit boundary, bit size 7	Error	13.8	Error
ss702	Time for accessing a packed component spanning a storage unit boundary, bit size 11	Error	13.7	Error
ss707	Time for accessing a packed component spanning a storage unit boundary, bit size 15	Error	13.8	Error
ss712	Time for accessing a packed component spanning a storage unit boundary, bit size 16	Error	9.0	Error
ss688	Time for accessing a packed component not spanning a storage unit boundary, bit size 3	Error	13.8	Error
ss693	Time for accessing a packed component not spanning a storage unit boundary, bit size 5	Error	14.1	Error
ss698	Time for accessing a packed component not spanning a storage unit boundary, bit size 7	Error	14.2	Error
ss703	Time for accessing a packed component not spanning a storage unit boundary, bit size 11	Error	13.7	Error
ss708	Time for accessing a packed component not spanning a storage unit boundary, bit size 15	Error	14.2	Error
ss713	Time for accessing a packed component not spanning a storage unit boundary, bit size 16	Error	9.0	Error
ss689	Time for both spanning and nonspanning accesses, bit size 3	Error	510.7	Error
ss694	Time for both spanning and nonspanning accesses, bit size 5	Error	511.6	Error
ss699	Time for both spanning and nonspanning accesses, bit size 7	Error	512.7	Error
ss704	Time for both spanning and nonspanning accesses, bit size 11	Error	513.4	Error
ss709	Time for both spanning and nonspanning accesses, bit size 15	Error	516.2	Error
ss714	Time for both spanning and nonspanning accesses, bit size 16	Error	343.8	Error
ss690	Time for allocating a unpacked array component to a packed array component, bit size 3	Error	217.3	Error
ss695	Time for allocating a unpacked array component to a packed array component, bit size 5	Error	218.0	Error

TABLE 25. SIZE Representation Clause Benchmarks (Continued)

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss700	Time for allocating a unpacked array component to a packed array component, bit size 7	Error	219.0	Error
ss705	Time for allocating a unpacked array component to a packed array component, bit size 11	Error	220.4	Error
ss710	Time for allocating a unpacked array component to a packed array component, bit size 15	Error	223.0	Error
ss715	Time for allocating a unpacked array component to a packed array component, bit size 16	Error	120.2	Error
ss691	Time for simple assignment to a packed array component, bit size 3	Error	10.5	Error
ss696	Time for simple assignment to a packed array component, bit size 5	Error	10.7	Error
ss701	Time for simple assignment to a packed array component, bit size 7	Error	10.4	Error
ss706	Time for simple assignment to a packed array component, bit size 11	Error	10.7	Error
ss711	Time for simple assignment to a packed array component, bit size 15	Error	10.4	Error
ss716	Time for simple assignment to a packed array component, bit size 16	Error	7.6	Error

TABLE 26. Record Representation Clause Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss717	Time to assign the left justified field of a packed boolean record to the NOT of its value	Error	1.6	4.1
ss718	Time to assign the next to left justified field of a packed boolean record to the NOT of its value	Error	1.6	4.6
ss719	Time to assign the next to right justified field of a packed boolean record to the NOT of its value	Error	1.5	6.8
ss720	Time to assign the next to right justified field of a packed boolean record to the NOT of its value	Error	1.6	6.7
ss724	Specify integer record using record representation clause and check for fields on storage_unit boundary	Error	1.4	2.8
ss725	Specify integer record using record representation clause and check for fields NOT on storage_unit boundary	Error	1.5	2.8

TABLE 27. Attributes

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss730	Address attribute of a subroutine	1.0	1.1	1.9
ss731	Address attribute of a package object	1.7	1.7	1.8
ss732	Address attribute of a dynamic object	1.6	1.5	1.7
ss734	Test SIZE attribute of a package object	2.1	2.1	2.9
ss735	Test SIZE attribute of a dynamic object	1.8	1.8	2.9
ss736	Test POSITION attribute for a record component	1.8	1.8	1.8
ss737	Test FIRST_BIT attribute for a record component	0.6	1.0	0.9
ss738	Test LAST_BIT attribute for a record component	0.6	0.6	0.8
ss739	Test STORAGE_TYPE for a access type	0.9	0.8	0.7
ss740	Test STORAGE_TYPE for a task type	0.9	0.9	0.9

TABLE 28. Unchecked_Conversion

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss259	Convert an array of real with bounds given by an enumeration type to array of real with bounds given by literal range	4.8	5.9	Error
ss500	Time for conversion to int from packed boolean array, AND operator on packed boolean arrays	10.8	3.9	8.6
ss501	Time for unchecked conversions between int and packed array of boolean	9.4	4.8	9.5
ss502	Unchecked conversion between INT and Boolean array, use overloaded OR and AND on INT as packed boolean operator	19.9	8.7	18.8
ss506	Unchecked conversion between INT and Boolean array, AND boolean array operator	10.9	4.7	8.5

TABLE 29. Interrupt Handling

Name	Description
int_1	Time simple interrupt. Raise interrupt and test a flag which is set by the handler. The interrupt task is initiated from the main task.
int_2	Task switching. An interrupt enables a task with a higher priority than the task which was running when the interrupt occurred. After the interrupt has been serviced, the higher priority task, not the one running when the interrupt occurred, will be scheduled.
int_3	In this test, several tasks are placed in a delay queue to test scheduling overheads associated with restarting execution after processing an interrupt. Timing of a simple interrupt is complicated by having several tasks on a wait queue.
int_4	There are several runnable tasks eligible at all times. These tasks have a lower priority that the task performing the null timing loop, and should not be executed.
int_5	Exception is raised within the rendezvous of the interrupt entry call. This problem tests for the performance impact of raising exceptions inside the rendezvous.
int_7	Int_7 tests the response time when an interrupt occurs during an interrupt handler. As the interrupt tasks have the same priority, this test will also check whether an interrupt will override an interrupt handler. The LRM is not clear in specifying that interrupt tasks must have priorities. It is permissible for all interrupts to be treated the same and not preempt each other
int_8	Int_8 tests the response time when an interrupt occurs during an interrupt handler. Int_8 is similar to Int_7 except the second interrupt task has a higher priority than the executing interrupt. This test will determine whether priorities are recognized by the handler.
int_9	Int_9 tests the response time when an interrupt occurs during an interrupt handler. Int_8 is similar to Int_7 except the second interrupt task has a higher priority than the executing interrupt. This test will determine whether priorities are recognized by the handler.

TABLE 30. Pragmas

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss444	Effect of suppressing division_check and overflow_check on floating point division	23.0	24.9	17.1
ss445	Effect of suppressing division_check and overflow_check on integer division	3.7	3.7	5.5
ss446	Effect of suppressing division_check and overflow_check on integer MOD operation	4.5	4.6	16.0
ss447	Effect of suppressing division_check and overflow_check on integer REM operation	3.8	4.1	8.5

TABLE 31. Subprogram Overhead Tests

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss36	Time for a simple procedure with no parameters, call to library scope procedure, body is null	2.2	2.1	7.9
ss37	Simple procedure with one IN OUT floating point parameters, declared in external library unit, body is null	3.6	3.7	8.7
ss38	Simple procedure with two IN OUT floating point parameters, declared in external library unit, body is null	5.3	6.8	8.9
ss39	Simple procedure with three IN OUT floating point parameters, declared in external library unit, body is null	5.1	8.3	7.6
ss141	Local function call with two real parameters	11.7	12.3	12.4
ss142	Local INLINE function with two real parameters	8.6	27.7	7.9
ss143	Local function call where actual parameter contains another function call	20.7	24.8	11.6
ss148	Inline generic procedure on strings	32.1	10.4	8.9
ss150	Inline generic procedure on floating point scalar	6.5	38.0	6.7
ss248	Procedure call with one out and in mode real parameters	14.8	11.4	6.7
ss419	Procedure call with a scalar parameter that is an element of a dynamically sized array	5.2	5.9	7.8
ss420	Procedure call with a scalar parameter that is an element of a dynamically sized array	5.3	4.5	8.9
ss478	Call a procedure which is a generic formal parameter	2.0	3.1	4.7
ss613	Procedure call with unconstrained record parameter	3.9	3.3	9.0
ss614	Procedure call with unconstrained array type	6.6	5.3	9.7
ss615	Procedure call with constrained record parameter	4.1	2.8	8.9

TABLE 32. Subprogram Overhead Tests (Continued)

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
ss616	Procedure call with unconstrained record parameter, suppression specified	4.3	3.9	7.9
ss617	Procedure call with unconstrained array type, suppression specified	7.2	6.8	10.0
ss618	Procedure call with constrained record parameter, suppression specified	4.0	4.5	9.6
ss621	Call to generic, non-inline function	16.3	14.9	8.9
ss622	Call to generic, inline function	16.2	16.0	11.7
ss623	Call to generic, non-inline function, might be automatically inlined	13.8	13.1	8.0
ss624	Call to generic, inline function	14.1	12.2	7.9
ss625	Call to local generic, inline function	10.7	26.7	11.8
ss626	Call to local generic, NOT inline function	14.3	10.8	11.8
ss633	Call to inline function in separate package	12.1	13.2	9.7

TABLE 33. Real-time Paradigms

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
task25	Bounded buffer with nonscalar parameter (string of length 10)	6140009.00	5333357.00	3120007.0
task47	Bounded buffer with scalar parameter	15552.1	15449.0	8999.7

TABLE 34. Composite Benchmarks

Time in Microseconds

Name	Description	HP3.25	HP4.35	Verdix
arti	Avionics application study example	1783.0	1647.9	1567.8
ew	Electronic warfare application feasibility study	Error	690.1	567.3
kalman	Application study, kalman filter	4702.8	3294.0	2890.9
simulate	Simulation application study	871.8	951.50	678.9

Appendix B: ACEC Benchmarks From a Real-time Perspective

The ACEC benchmarking suite was primarily developed for evaluating the runtime performance of Ada compilers. Since the suite is comparatively new, not many organizations have run the ACEC with the intent of evaluating compilers. Being one of the first organizations outside of the Air Force that evaluated and ran the ACEC on different compiler implementations, a number of problems about the ACEC have surfaced. These problems are discussed below.

1. *INCLUDE Preprocessor:* The ACEC timing code loop consists of four (4) code files which are incorporated into the source by a preprocessor (INCLUDE) which supports text inclusion. INCLUDE is written as an Ada program and has to be modified on systems which do not support the concept of file suffixes as well as Ada compilers which require Ada programs to have a suffix ".a" instead of ".ada".

   However, the biggest problem with INCLUDE is that for cross-compilers the INCLUDE command runs on the target and for bare machine implementations of Ada that may not support a file system, the INCLUDE preprocessor will not work. However, if the target does support a file system, it takes a very long time for a single ACEC test to run through the INCLUDE preprocessor (on the Verdix cross compiler, it took nearly 30 minutes to run a file through the INCLUDE preprocessor).

2. *Large Number of Tests:* The ACEC suite consists of over 1000 tests and many (~25%) of these tests do not provide any useful information about compiler evaluation. Also, to compile and run the whole suite can take somewhere from 3 to 5 days. Once the tests have been run, it is extremely difficult for an organization to select the relevant tests (from a real-time perspective) and then interpret the results in order to do a thorough compiler evaluation. Substantial effort may be required on the part of an organization to run the ACEC and interpret the results to perform compiler evaluation. An organization may not have the time and resources to run the ACEC and then interpret the results for compiler evaluation.

3. *MEDIAN Analysis Tool:* The primary focus of the ACEC is on comparing performance data between different compilation systems rather than on studying the results of one particular system. The MEDIAN analysis tool isolates problems where a system does unusually well or poorly. The problem with this kind of analysis is that if for example two compilers are tested and both compilers have nearly similar rendezvous timings, the timings may not meet the real-time requirements. The MEDIAN analysis tool will not isolate this problem, and an organization will have to analyze the individual test cases to gain more information. If an organization just depends on the output of MEDIAN, it is quite possible that problems with the runtime performance of

real-time Ada features may not be detected. Also, the summary statistics attach equal weight to all the problems and although one compiler may have a better summary data, it does not imply that it is better than the other compiler in areas that are of particular interest for an application. Also, it is a tremendous amount of effort to analyze the statistical numbers produced for all the problems (> 1000).

4. *Ada Application Code:* The ACEC contains code from some applications that were developed in organizations using Ada. This code is not very useful and does not provide any relevant information for compiler evaluation for real-time systems. The code of these composite benchmarks is not designed using good software engineering practices and it appears that Fortran code may have just been converted to Ada. Also, there is no information to interpret the results produced by these benchmarks and it is extremely difficult to go through the code and figure out what the results of the benchmarks mean.

5. *Interpretation of Results:* In many instances, it is not clear as to why a certain test has been provided and what is the purpose of running that test. Also, in many cases, the benchmark code has to be read in order to interpret the results of the output. More information needs to be provided in order to help interpret the results produced by running the benchmarks.

6. *Large Number of Iterations:* The ACEC tests dynamically compute the number of iterations necessary to obtain measurement within a specified accuracy. For many tasking tests, it is quite possible that the number of iterations is rather large, so that the benchmark could run out of memory while executing the required number of iterations (due to memory being allocated as more tasks are activated). This was experienced in the case of the Verdix Ada compiler targeted to the Motorola 68020 bare machine with 2 megabytes of RAM.