

AD-A236 716



ADA COMPILER EVALUATION CAPABILITY

Reader's Guide, Release 2.0

Thomas Leavitt
Kermit Terrell

Boeing Military Airplanes
Post Office Box 7730
Wichita KS

May 1991



Interim Report

Approved for public release; distribution is unlimited.

91-01853



AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

91 6 11 132

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

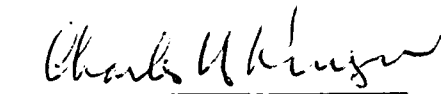
This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.


RAYMOND SZYMANSKI
Project Engineer

25 March 1991
Date

FOR THE COMMANDER



3 APR 1991
Date

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAF, WPAFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

May 1991

Interim

Ada Compiler Evaluation Capability
Reader's Guide, Release 2.0

C-F33615-86-C-1059

PE-63756D

PR-2853

TA-01

WU-03

Thomas Leavitt
Kermit Terrell

Boeing Military Airplanes
Post Office Box 7730
Wichita KS

Raymond Szymanski (513) 255-3947
Avionics Directorate (WL/AAAF)
Wright Laboratory
Wright-Patterson, AFB, OH 45433-6543

WL-TR-91-1041

Approved for Public Release; Distribution is unlimited

The Ada Compiler Evaluation Capability (ACEC) is a set of over 1500 performance and usability tests used to assess the quality of Ada compilers. The ACEC also provides statistical analysis tools to assist in analyzing the results generated by the ACEC. The ACEC is documented through three major documents; the ACEC Reader's Guide, the ACEC User's Guide and the ACEC Version Description Document.

This document, the ACEC Reader's Guide, describes how ACEC users can interpret the results of executing the ACEC, the statistical significance of the numbers produced, the organization of the test suite, how to find particular language features and/or specific optimizations, and how to submit error reports and change requests.

Ada, Compiler, Evaluation, ACEC
Metrics, Evaluation & Validation Project

156

Un

Un

Un

DTIC users

LIMITATIONS

This document is controlled by the Boeing Military Airplanes (BMA) Software and Languages Organization. All revisions to this document shall be approved by the above organization prior to release.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



ABSTRACT

This Reader's Guide for the Ada Compiler Evaluation Capability describes how end users can interpret the results of executing the benchmark test suite, the statistical significance of the numbers produced, the organization of the test suite, how to find particular language features and/or specific optimizations, and how to submit error reports and change requests.

ACEC
Reader's Guide

Contents

1	SCOPE	10
1.1	IDENTIFICATION	10
1.2	PURPOSE	10
1.3	INTRODUCTION	10
2	APPLICABLE DOCUMENTS	12
2.1	GOVERNMENT DOCUMENTS	12
2.2	NON-GOVERNMENT DOCUMENTS	12
3	ORGANIZATION OF THE TEST SUITE	14
3.1	SCOPE OF THE ACEC	14
3.1.1	What Is And Is Not Covered By The ACEC	14
3.2	EXECUTION TIME EFFICIENCY	17
3.2.1	Individual Language Features	18
3.2.2	Pragmas	21
3.2.3	Optimizations	22
3.2.3.1	Classical Optimizing Techniques	22
3.2.3.1.1	Common Subexpression Elimination	22
3.2.3.1.2	Folding	23
3.2.3.1.3	Loop Invariant Motion	23
3.2.3.1.4	Strength Reduction	23
3.2.3.1.5	Dead Code Elimination	23
3.2.3.1.6	Register Allocation	23
3.2.3.1.7	Loop Interchange	24
3.2.3.1.8	Loop Fusion	24
3.2.3.1.9	Test Merging	24
3.2.3.1.10	Boolean Expression Optimization	24
3.2.3.1.11	Algebraic Simplification	24
3.2.3.1.12	Order Of Expression Evaluation	25
3.2.3.1.13	Jump Tracing	25
3.2.3.1.14	Unreachable Code Elimination	25
3.2.3.1.15	Use of Machine Idioms	25
3.2.3.1.16	Packed Boolean Array Logical Operators	26

3.2.3.2	Effect of Pragmas	26
3.2.3.3	Static Elaboration	26
3.2.3.3.1	Aggregates	27
3.2.3.3.2	Tasks	27
3.2.3.4	Language Specific	28
3.2.3.4.1	Habermann-Nassi Transformation For Tasking	28
3.2.3.4.2	DELAY Statement	28
3.2.3.4.3	NULL statement	30
3.2.4	Performance Under Load	30
3.2.4.1	Task Loading	30
3.2.4.2	Levels of Nesting	30
3.2.4.3	Parameter Variation	31
3.2.4.4	Declarations	31
3.2.5	Tradeoffs	31
3.2.5.1	Design Issues	32
3.2.5.1.1	Order of Evaluation	32
3.2.5.1.2	Default vs Initialized Records	32
3.2.5.1.3	Order of Selection	32
3.2.5.1.4	Scope of Usage	32
3.2.5.1.5	LOOP Statements	33
3.2.5.1.6	CASE Statement	33
3.2.5.1.7	Subtypes	33
3.2.5.1.8	Generics	33
3.2.5.1.8.1	Text IO Elaboration Time	34
3.2.5.1.9	Library Subunits	34
3.2.5.1.10	Exceptions	34
3.2.5.2	Context Variation	35
3.2.5.2.1	Different Coding Styles	35
3.2.6	Operating System Efficiency	35
3.2.6.1	Tasking	35
3.2.6.2	Exception Handling	35
3.2.6.3	I/O	35
3.2.6.4	Memory Management	42
3.2.6.4.1	Implicit Storage Reclamation	42
3.2.6.5	Elaboration	43
3.2.6.6	Runtime Checks	44
3.2.6.6.1	Elaboration Checks	44
3.2.7	Application Profile Tests	45

3.2.7.1	Classical Benchmark Programs	45
3.2.7.2	Ada in Practice	45
3.2.7.3	Ideal Ada	47
3.3	CODE SIZE EFFICIENCY	47
3.3.1	Code Expansion Size	47
3.3.2	Runtime System Size	48
3.4	COMPILE TIME EFFICIENCY	48
3.5	TESTS FOR EXISTENCE OF LANGUAGE FEATURES	49
3.6	USABILITY	50
3.6.1	Symbolic Debugger Assessor	50
3.6.2	Program Library System Assessor	52
3.6.3	Diagnostic Assessor	53
3.7	CAPACITY TESTS	54
3.8	SUMMARY	55
3.9	SIMPLE STATEMENT TESTS	55
4	HOW TO INTERPRET THE OUTPUT OF THE ACEC	57
4.1	OPERATIONAL SOFTWARE OUTPUT	57
4.1.1	Results File	58
4.2	MEDIAN OUTPUT	62
4.2.1	Raw Data	62
4.2.2	Histograms	63
4.2.3	Residual Data	70
4.2.4	Statistical Summaries	71
4.2.5	Residual Data Summary	73
4.2.6	Problem Name Outliers	74
4.2.7	System Summary	75
4.3	SINGLE SYSTEM ANALYSIS	76
4.3.1	Table of Contents	78
4.3.2	Main Report	80
4.3.2.1	Multiple Comparisons	80
4.3.2.2	Statistically Significant Differences	82
4.3.2.3	Binary Comparisons	83
4.3.2.4	Ancillary Data	85
4.3.2.5	Failure Analysis Report	85
4.3.2.6	Code Size Report	86
4.3.2.7	Compile Speed Report	86
4.3.3	Report Summary	86
4.3.4	Missing Data Report	88

5	DETAILS OF TIMING MEASUREMENTS	90
5.1	REQUIREMENTS OF THE TIMING LOOP CODE	91
5.2	A PRIORI ERROR BOUND	93
5.2.1	Random Errors	94
5.2.2	Systematic Errors	95
5.3	HOW TEST PROBLEMS ARE MEASURED	109
5.4	TIMING TEST PROBLEMS	112
5.4.1	Clock Vernier	113
5.5	CODE EXPANSION MEASUREMENT	120
5.6	CORRECTNESS OF TEST PROBLEMS	120
5.6.1	Validity of Test Problems	120
5.6.2	Correct Translation of Test Problems	121
5.6.3	Satisfaction of Intent	121
6	STATISTICAL BACKGROUND FOR MEDIAN	124
6.1	MATHEMATICAL MODEL	124
6.2	STATISTICAL MODEL	126
6.2.1	Median Polish	129
6.2.2	Outliers	130
6.2.3	Optimization Analysis	131
6.2.4	Analysis of Results	133
6.2.5	Summary	135
6.3	TIMING DATA	137
6.3.1	Extrapolation of Timing Data	138
6.3.2	Tuning for Implementation Dependencies	140
6.4	CODE EXPANSION	141
6.5	RTS SIZE	142
6.6	COMPILATION TIME	143
7	VERSION DESCRIPTION DOCUMENT (VDD)	146
8	SIGNIFICANCE OF ERRONEOUS TESTS	147
9	UPWARD COMPATIBILITY OF THE TEST SUITE	148
10	ACEC USER FEEDBACK	148
10.1	HOW TO REQUEST CHANGES	149
10.2	HOW TO REPORT ERRORS	151

11 NOTES

11.1

ABBREVIATIONS, ACRONYMS, AND THEIR MEANINGS . . 153

List of Figures

1	Sample Input to FORMAT - Generated by Test Programs	59
2	Example of a Partial Table of Contents Page	79
3	Multiple Comparison Table Example	81
4	Example of Main Report : Paired Comparisons Page	84
5	Example of a Report Summary	87
6	Example of a Partial Missing Data Report	89
7	Timing Loop Template	111
8	Clock Vernier	114

1 SCOPE

This section identifies the Ada Compiler Evaluation Capability (ACEC) product, states its purpose, and summarizes the purpose and contents of this Reader's Guide.

1.1 IDENTIFICATION

This is the Reader's Guide for the ACEC.

1.2 PURPOSE

The purpose of this guide is to instruct end users on the interpretation of the results of the ACEC. This includes the timing/sizing measurements from executing the performance test suite, the outputs from the performance analysis tools (MEDIAN and SINGLE SYSTEM ANALYSIS), the debugger assessor, the library system assessor and the diagnostic assessor. It describes the organization of the test suite, the output of the ACEC, the details of the timing measurements, the statistical background for the analysis program, the contents of the Version Description Document, how to request changes, and how to report errors in the ACEC.

Instructions for installing and executing the ACEC are covered in the User's Guide. If one individual will be responsible for both running the ACEC and interpreting the results, the User's Guide should be read first.

After reading this guide, the meaning and significance of the various reports produced by the ACEC will be clear.

1.3 INTRODUCTION

The ACEC Software Product consists of a portable test suite and support tools. The test programs will, when executed, measure the execution time and space on a target system with a systematically constructed set of Ada examples. The support tools assist the ACEC user in executing the test suite and analyzing the results obtained. The ACEC also contains units (both erroneous and non-erroneous) to explore a system's diagnostic messages (errors and warnings); test programs and scenarios to evaluate a symbolic debugger; and compilation units and command scripts to evaluate an Ada program library system's capabilities.

Because the ACEC is based on a test suite, it will not be useful in situations where no actual compiler is available for evaluation. The ACEC cannot be used to assess implementors' claims of what their system *will* do. If an implementor has a version which is operational, the user may be able to partially assess some vendor claims. If a system does no optimizations at all, and the ACEC confirms that no optimizations are being performed, a user might believe that there is room for improvement in the system.

This document is intended for users who want to interpret the output of the test suite and tools. "Readers" are persons or organizations who want to interpret the raw data collected by executing the ACEC benchmark test suite, understand the statistical significance of the data and the analysis performed on it, and learn how to use the ACEC results to answer specific questions about various implementations. Different classes of readers are interested in different kinds of information about Ada compilation systems, and will want the ACEC to provide them with different types of analysis.

A reader concerned with selecting one system with the best performance from several candidates has different concerns than a reader who is managing a project and wants to know what performance changes were made in a new compiler release. Implementors concerned with enhancing the performance of their product have a different set of concerns, as do developers of portable software packages. All should be able to extract useful information from the ACEC.

The User's Guide discusses the procedures to compile and execute the test programs and the support tools. It discusses potential problems and adaptations which may need to be performed. For information on how to install and run the ACEC, readers should consult that document. That guide describes the use of the support tools INCLUDE, FORMAT, MED DATA CONSTRUCTOR, MEDIAN, and SSA, and the steps which must be taken to produce output. This guide assumes that the outputs have been produced, and describes the format and significance of the output.

The ACEC is developed for uniprocessor, uniprogramming target systems.

2 APPLICABLE DOCUMENTS

The following documents are referenced in this guide.

2.1 GOVERNMENT DOCUMENTS

MIL-STD-1815A Reference Manual for the Ada Programming Language (LRM)

2.2 NON-GOVERNMENT DOCUMENTS

D500-12470-1 Ada Compiler Evaluation Capability (ACEC)
Technical Operating Report (TOR)
User's Guide
Boeing Military Airplanes
P.O. Box 7730
Wichita, Kansas

D500-12472-1 Ada Compiler Evaluation Capability (ACEC)
Version Description Document (VDD)
Boeing Military Airplanes

"An Empirical Study of FORTRAN Programs" by D. Knuth in
Software: Practice and Experience, Volume 1, Number 2,
1971.

"Re-evaluation of RISC I" by J. L. Heath in Computer
Architecture News, Volume 12, Number 1, March 1984.

"A Performance Evaluation of The Intel iAPX 432" by Hansen,
Linton, Mayo, Murphy and Patterson in Computer Architecture
News, Volume 10, Number 4, June 1982.

"How Not to Lie with Statistics: The Correct Way to
Summarize Benchmark Results" by P. Fleming and J. Wallace
in CACM, Volume 29, Number 3.

Applications, Basics and Computation of Exploratory Data Analysis, by Velleman and Hoaglin, Durbery Press, 1981.

ALGOL60 Compilation and Assessment, B. Wichmann, Academic Press, 1973.

Low Level Self-Measurement in Computers, R. Koster, School of Engineering and Applied Science, University of California Los Angeles, Report Number 69-57, 1969.

Compilers: Principles, Techniques, and Tools, A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1986.

Optimizing Supercompilers for Supercomputers, M. Wolfe, University of Illinois at Urbana-Champaign, 1982

Introduction To the Theory of Statistics, A. Mood and F. Graybill, McGraw-Hill, 1963

"Task Management in Ada — A Critical Evaluation for Real-Time Multiprocessors," by E. Roberts, A. Evans, C. Morgan, and E. Clarke, Software:Practice and Experience, Volume 11, Number 10, October 1981.

Understanding Robust and Exploratory Data Analysis, by D. Hoaglin, R. Mosteller, and J. Tukey, John Wiley & Sons, 1983

3 ORGANIZATION OF THE TEST SUITE

The test suite contains a large number of test programs. A test program may contain more than one test problem. This is helpful in executing the ACEC on an embedded target where downloading from the host can take a large amount of time. The time to run the whole test suite can be significantly reduced if more than one problem is downloaded at a time. This can also reduce the total compilation time, since several programs are combined into one which can share various pieces of logic (such as instantiations of TEXT_IO).

Each test problem is embedded in a template which, when executed, will report the execution time and the code expansion space for the problem. The template is discussed in Section 5.3.

3.1 SCOPE OF THE ACEC

The emphasis of the ACEC is on determining the execution performance on a target system, both for speed and memory space. 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, consistent with the need to test all Ada language features of interest to Mission Critical Computer Resource (MCCR) applications, which are the class of applications of special interest to the ACEC.
- 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*.

3.1.1 What Is And Is Not Covered By The ACEC

The best performance predictor for any particular program is the program itself. Only the final program will enable users to determine whether their implementation will satisfy their performance requirements. That being said, it must be recognized that most projects will want to use the ACEC before their final program is available, so they can select an appropriate compilation system (compiler and target processor).

Even when a working version of the final program is available, users may find that its performance is not acceptable and want information to assist them in enhancing it. Performance analyzers (also called program profilers) are helpful in isolating areas of a program which account for most of the execution time of a program. However, this does not tell a programmer

whether or not the construction used within the program's "hot spots" might be faster if coded using different language constructions. If a project is concerned with selecting which of several implementations to use on a project, this approach requires the final program be ready for testing before the selection of the compilation system. And even if a "very similar" application is available, the effort required to adapt it to each candidate system may be large, involving learning about system dependencies. A compiler implementor would not receive any information about which language features of their system are particularly slow (or fast) relative to other implementations, which they could use to help determine areas where performance improvements should be possible. Also, with only a set of final application programs, programmers have no initial guidance as to which language features or coding techniques they should try to avoid on a particular implementation. The ACEC aims to provide such information. To do so, the ACEC includes a set of tests which provides a broad scope of coverage of various features of the language.

The ACEC is organized as a set of essentially independent test problems, assessors, and analysis tools. It is easy for users to insert additional test problems that they find of particular interest. The analysis tools will process user defined test problems as easily as the ACEC developed problems. Users should consult Section 5 and the User's Guide, Section "CONSIDERATIONS FOR CODING ADDITIONAL TESTS", for advice on test construction.

The issues addressed and not addressed by the ACEC are:

- ADDRESSED

- Execution time
- Code expansion size
- Compile time
 - a side effect of compiling programs developed to test execution time.
- Diagnostics
- Ada program library management systems
- Symbolic debuggers

- NOT ADDRESSED

- Capacity limits
 - There are not specific tests to systematically probe for capacity limits. However, some of the ACEC test programs are sufficiently large that they have exceeded limits on some systems. The library management assessor does exercise the capacity limits of a system's library capabilities.

- Cost
- Adaptability to a special environment

This includes the ability to modify the Runtime System (RTS) to fit a special *target* configuration which is important to many MCCR projects. Modification to an RTS brings up validation and revalidation questions, and extensive modifications may require revalidation.

- Presence of support tools

A cross reference listing; variable set/use lists; management support tools; compilation order dependency tools; how well the host Ada compilation system works with configuration management systems; etc.; are all tools which can make a system easier to work with and both speed up the program development process, ease the maintenance task, and generally support the program life-cycle.

There are some target processors which have not been explicitly anticipated in the suite of test problems, and a reader may ask how well unconventional target processors will be evaluated, or what extensions to the ACEC would be needed to adequately cover them.

- Vector processors. There are test problems which can be vectorized, (Livermore). However, the machine idioms do not completely cover all the cases of interest. In particular, each *vector processor architecture* places particular constraints on what sequence of operations can and can not be processed with vector instructions. For example: vector registers may be limited in size; the memory span between elements which can be processed in vector mode may be restricted (to 1 in some designs); the efficiency of vector mode instructions on short vectors may be such that scalar processing is more efficient; some conditional processing in vector mode may be supported on some targets; special advice may have to be given to Ada compilers to permit vectorizing (particularly with respect to numeric exception processing in vector mode); and special vectorized library functions may be available (and necessary) to exploit the vector processing capabilities.
- Very Long Instruction Word (VLIW) machine architectures. This class of machines have different machine idioms than others. It is possible that some of the specific optimization tests designed for other target machines will be falsely reported as being performed. For example, there are several common subexpression tests where an optimizing compiler may reduce the operation count by generating fewer instructions and taking less time than a version of the problem written as two statements with the common subexpression merged "by hand;" a VLIW processor may use several parallel instruction units to perform more operations for the "unoptimized" test problem, but still complete it in the same time.

Some care should be used in interpreting results on such a target. There are few Ada compilers available for such targets, so experience is limited.

- Reduced Instruction Set Computer (RISC). On these processors, the only real difficulty is interpreting the test problems which probe for machine idioms. If there is no Increment Memory instruction on a target, test problems which permit its use are not directly applicable to the target machine. However, the time to increment a counter, or zero memory, or perform other operations which some machines have developed *special idioms* to support efficiently, are still of interest — adding to or zeroing a counter is something which many programs need to do, whether it can be done in one machine instruction or not. The time (and space) it takes to perform the operation is the topic of real interest, not the machine instructions used to implement the operation. Special, idiomatic, instructions were generated for machines because the designers believed that the instructions would be useful. However, for any particular target Instruction Set Architecture (ISA), a user may be especially interested in how well the target machine instructions are utilized: Are available idioms exploited? To answer this question, specific test problems need to be executed, and if not present, may need to be developed.
- Multicomputer. On a network of independent processors, with or without shared memory, it may be possible to process tasks in parallel for faster execution. In particular, it may be possible for a system to allocate separate Ada tasks to run in parallel on different processors. This may give interesting results, but when comparing performance between a uniprocessor system and a multicomputer, readers will need to understand the target configuration and the number of processors.
- 1750A Extended Memory. Access to MIL STD 1750A extended memory is being provided by different implementors in different, incompatible ways. It is not the intention of the ACEC to explore the performance of the different implementation dependent extensions to provide access to extended memory on the 1750A. If the ACEC tests are run as written, a baseline will be developed which can be used to compare systems. Projects requiring extended memory will generally need to modify the tests to match the extension needed for each implementation they test.

3.2 EXECUTION TIME EFFICIENCY

Determining the execution speed of various problems was the major factor in the design of the ACEC.

3.2.1 Individual Language Features

There are test problems for all major Ada language features. One test problem will rarely be sufficient to accurately characterize the performance of a language feature. Optimizing compilers can generate different machine code for the same source text statement, depending on the context in which it occurs. 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.

For example, consider the simple assignment statement " $i := j;$ " where " i " and " j " are both integers:

1. If " j " is a loop invariant in a loop surrounding the assignment, the statement may be moved out of the loop.
2. If " i " is not referenced after being assigned to before being reassigned or deallocated, dead code elimination may eliminate the statement.
3. If the value of " j " may be known to be some constant based on statements executed before this assignment, it may be possible to use a special store constant or clear (if the value is known to be zero) instruction on the target machine.
4. If an assignment was made to " j " earlier in the basic block leading to this assignment, it may be available in a register, permitting the compiler code generator to avoid a redundant load from memory of " j ."
5. If the value of " i " is needed later in the sequence of statements following the assignment, it may be possible to defer the store into " i ."
6. Establishing addressability to " i " and/or " j " may require the setup of a base register, or it may be possible that the required addressability has been established by other statements. This may be a common requirement if the variable is declared with intermediate scope — neither local nor library scope.
7. If " i " has range constraints, various amounts of data flow analysis may be able to determine if " j " must satisfy the constraint or if explicit tests need to be generated. The performance in general will depend on whether "pragma suppress" has been specified or not.
8. If the assignment statement is unreachable, occurring inside the **THEN** clause of an **IF** statement determinable at compile time to be false, or directly after a **GOTO**, **RAISE**, **EXIT**, or **RETURN** statement without a label on the statement, or within a **FOR** loop whose range is determinable to be null at compile time, or within a **WHILE** loop

determinable at compile time to have a false condition, or in a CASE alternative which is determinable at compile time not to be selectable, no code need be generated for it at all.

9. If "i" has range constraints and "j" is known at compile time by data flow analysis to be outside the permissible range, the compiler may simply raise an unconditional `constraint_error`.

Some language features are tested with a corresponding systematic variation of contexts to expose the range of performance. Examples include: subprogram calling; task rendezvous; exception processing; arithmetic expressions; and I/O operations. Although many embedded targets will not support file systems, many Command and Control MCCR applications make intensive use of file systems and the performance of I/O operations is critical to their application performance.

There is a set of test problems for "pragma pack" which measure both time and space. Some packing methods do not 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. There are test problems which access a packed component of a record which would span a storage unit boundary (if densely packed), and others that access components which are left and right justified in a storage unit. For justified components, machine idioms may be able to access the components more efficiently than for general alignments. These tests will not require tailoring to the storage unit sizes on the system under test; however, a test problem which forces a field to span a boundary on one system may not do so on all target systems. Although which component spans a boundary is dependent on the implementation storage unit size, the computation to identify the component can be performed in an implementation independent manner using the named number `SYSTEM.STORAGE_UNIT`. There will be test problems for the storage unit sizes in common usage: 8, 12, 16, 24, 32, 48, 60, and 64 bits. In addition to measuring the time to perform the test problems accessing packed objects, these test problems will use the representation attribute, `X'SIZE` (LRM 13.7.2) to determine the actual bit size of the objects and compare this with the predetermined minimum possible bit size for the object. These sizes will be printed for information: they show the degree of packing the system under test performs.

Major language features are defined to be the features which are expected to require execution time and to have a reasonable expectation of portability. This includes all syntactic constructions except those listed below, by LRM Section number:

1. LRM 2.4.2 Based Literals.

The ACEC assumes that programs will be compiled, and literal values represented in an internal form at execution time. The format of the external representation of a literal

value should not make any difference in execution time or space in non-source-interpretive systems.

2. LRM 2.7 Comments.

The presence of comments in test problems should not make any difference in execution time or space in non-source-interpretive systems.

3. LRM 2.10 Replacements of Characters.

The use of alternate character forms should not make any difference in the execution time or space in non-source-interpretive systems.

4. LRM 13.5 Address clauses.

The ACEC will not force users to make system dependent *modifications to test references* to variables that are bound to fixed locations by ADDRESS clauses. Tying tasks to interrupts is a special case which is tested, because the performance characteristics are important and likely to be highly variable. For portability, the ACEC does not include tests which use an ADDRESS clause to map a variable to a fixed location.

5. LRM 13.8 Machine Code Insertions.

Test problems for machine code insertion are fundamentally tests of the underlying target hardware, rather than of any property of the Ada system. Designing a portable test problem for different target machines would be awkward, and impossible to verify since each user would have to perform the adaptation to the target machine. Even for the same target machine, different compilers may use different subprogram linkage and register usage conventions making a valid code insertion for one system erroneous on another, some examples:

- One system may expect that all machine registers are preserved across machine code insertions, while on another it may be permissible to modify various register values — if one compiler leaves a **FOR** loop index in a register and a piece of inserted machine code modified the register, results could be peculiar.
- It may be possible to write a piece of machine code which, when placed within an exception handler for OTHERS, examines the runtime environment provided for debug support and determines the identification of the exception which is being processed and where it was originally raised (line number or subprogram name). This could be a very useful piece of code, but is not likely to be portable.
- The machine code to establish addressability to library units can differ between implementations — one system may allocate an object to a static (bound at link

time to a fixed address) location, where another does not bind it until the package is elaborated. A piece of machine code which assumes the first alternative is performed and loads the address of the object in one instruction into a register to manipulate it will fail totally if the actual address needs to be computed by adding the base address of a package or by following a level of indirection.

- The layout of records, including the degree of packing of fields, placement of discriminates, and the format and interpretation of descriptors for unconstrained type objects, may differ between implementations. Machine code written assuming one layout may be completely inappropriate for another.

It is possible that the text formats are incompatible between implementations: different mnemonics for the machine instructions, different placement of instruction fields, etc.

6. LRM 13.9 Interface to Other Languages.

Calls to other languages are implementation dependent and must be adapted to each system. There is one test problem, SS747, which users must adapt to each target, which will call on a null procedure coded in assembler. As such it is basically a test of interlanguage linkage conventions. This is by no means exhaustive.

This simple control linkage test will not answer many of the interesting questions users have about the performance of a multi-language program. Data structure layouts often differ between implementations, and special access functions may be necessary to reference data structures defined in one language when referenced from the other, or shared data may be restricted to scalar items. Measuring the performance of a simple control transfer will not expose the costs of such data access functions.

Ensuring comparability in interfacing with a null assembler procedure is not easy. On many target machines, there are different sets of linkage conventions, of decreasing generality and increasing efficiency. For example, on the DEC VAX the CALLS mechanism is more general, but slower than the JSB mechanism. In "real" applications which call assembler coded subprograms, the linkage convention used will depend on the details of the program design, and represents a major design decision.

3.2.2 Pragmas

There are certain predefined pragmas which are expected to have an impact on the execution time and space of a program. These include: CONTROLLED, INLINE, OPTIMIZE, PACK, PRIORITY, SHARED, and SUPPRESS. Others are concerned with presentation information (LIST and PAGE), and with information needed for the description of a target system

or pieces of one: ELABORATE, INTERFACE, MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME. There are test problems which explore the performance effects of specifying those pragmas in the first list.

3.2.3 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. These examples permit a reader to determine if the optimization is ever performed. The comparisons between systems performed by MEDIAN will not distinguish between the case where all tested systems perform the optimization and the case where none did. The SSA report on optimizations should show which optimizations are performed. These test problems will not be amenable to optimizations other than the one being studied.

The set of Ada language features which, when used in various contexts might affect the performance (time and space) of the generated code is much too large to consider exhaustively enumerating all permutations of basic features. It is necessary to be selective in the construction of the test problem suite. Sets of related problems will be constructed with variations based on the use of language features expected to demonstrate the presence of specific optimizations.

3.2.3.1 Classical Optimizing Techniques

There are test problems which check for the presence of specific compiler optimizations. More detailed discussion can be found in texts on compiler writing, such as Compilers: Principles, Techniques, and Tools, by A. Aho, R. Sethi, and J. Ullman. The test suite contains problems for the listed techniques. The reader can refer to the Version Description Document Appendix VIII, "OPTIMIZATION TECHNIQUES" to find listings of techniques and the test problems which the techniques are particularly relevant to. The following subsections describe individual techniques.

3.2.3.1.1 Common Subexpression Elimination

This is the recognition that an expression previously evaluated need not be evaluated a second time as long the results of the first calculation are still available.

Although many compilers will recognize common subexpressions in some contexts, the better optimizing compilers do it in more contexts.

3.2.3.1.2 Folding

Folding is the performing of operations at compile time, including evaluation of arithmetic expressions with static operands (or operands whose values can be determined), and the folding of control structures (for example, no code needs to be generated for a statement of the form IF false THEN RAISE program error; END IF;).

3.2.3.1.3 Loop Invariant Motion

This is the moving of a statement (or computation) which is written within a loop, but which does not vary between iterations of the loop, to a place before the loop.

Greater than linear performance gains may be achieved through this technique, since instead of executing the moved code once per loop iteration, it will be executed only once. Some care should be exercised in applying this technique, since a loop which is normally not executed at all (for example, a WHILE loop which is typically initially false) might run slower with loop invariant motion than without it. This problem can usually be minimized by checking the condition that the loop may not be executed at all before evaluating the "moved" code.

3.2.3.1.4 Strength Reduction

This is a replacement of a *strong* operator with a *weaker*, and hopefully faster one. For example, multiply can be replaced by an add in the expression " $i * 2$ " resulting in the expression " $i + i$." A frequent and profitable application is the reduction of FOR loop indexes used as subscripts in arrays (which produce multiplication of the index by a constant reflecting the span between logically consecutive elements of one dimension of an array) with addition.

3.2.3.1.5 Dead Code Elimination

An assignment to a variable which is not referenced again before being either deassigned or deallocated is dead and need not be actually performed. While there are instances in which data is legitimately written to a memory location and never read (such as memory mapped IO), these cannot happen inside Ada unless the compiler knows that a variable is tied to a particular memory location by an ADDRESS clause (see LRM 13.5).

3.2.3.1.6 Register Allocation

On machines with multiple general purpose registers, the registers serve as fast temporaries which can save the values of expressions between simple statements. If used well, significant reductions in the number of instructions required to execute code are possible.

3.2.3.1.7 Loop Interchange

This is the process of switching inner and outer loops. It has a profound effect on the order of execution of statements, and can produce large performance gains.

The code can be transformed to contain more unit-stride array references. That is, the difference in memory addresses between references to an array on consecutive iterations is one. This can permit, or greatly enhance, operations on vector processors.

It can change the number of instructions on non-vector processors. For example, when an outer loop is performed 1000 times and an inner loop 5 times, there will be 1001 loop initiations (1000 for the inner loop and 1 for the outer loop). If the loops were switched, only 6 loop initiations will be required (5 for inner loop and 1 for outer loop).

Loop interchange may permit a reduction in paging on virtual memory machines by reducing the size of the working set.

Refer to Optimizing Supercompilers for Supercomputers by M. Wolf for more discussion.

3.2.3.1.8 Loop Fusion

This is the merging of loops with equivalent bounds. This can reduce the amount of loop overhead in a program.

3.2.3.1.9 Test Merging

This is the combination of tests, best demonstrated by example. Consider the code

```
IF i > 0 THEN ... ELSIF i = 0 ... ELSE ... END IF;
```

On most machines, the comparison for "i > 0" will set condition codes which can be retested for the condition "i = 0", that is the generated code could be: ..., compare, branch_less_equal to \$1, ... \$1: branch not equal \$2, ... \$2: ...

3.2.3.1.10 Boolean Expression Optimization

A boolean expression which has a static, or a derivable, operand can be simplified. An expression which simplified to "false AND i=j" can be further simplified into false. Demorgan's rule can be applied to help simplify boolean expressions.

3.2.3.1.11 Algebraic Simplification

There are various algebraic identities which can be used to simplify expressions. For example, it is not necessary to actually multiply by one or add zero.

3.2.3.1.12 Order Of Expression Evaluation

By evaluating expressions in a non-canonical order, it is often possible to compute the results faster. This can reduce the number of temporaries, permit special form (immediate operations), or permit register-memory operands rather than loading a value from memory followed by a register-register operand.

3.2.3.1.13 Jump Tracing

This is also known as "branch tracing."

A jump instruction which goes to another jump can be simplified into a single jump. Two consecutive jump instructions, the first conditional jump which skips over the second unconditional jump, can be merged into one conditional jump, using the inverse of the original condition, to the target of the original unconditional jump.

Programmers will not often write a **GOTO** which branches to another **GOTO**. However, a straightforward translation of conditional control structures often produces such sequences of branches.

3.2.3.1.14 Unreachable Code Elimination

Code which is not reachable can be eliminated. Statements following an unconditional **EXIT**, **RAISE**, **GOTO**, or **RETURN** cannot be reached and need not have any code included in the memory load. When the compiler can determine that an **IF** statement condition is always false, the **THEN** alternative need not have any code generated for it. Some may argue that since well written programs should not have this type of unreachable code, having a compiler which optimizes it is not very important. However, with optimizing compiler performing both inline expansions and generic instantiations, there are more conditional expressions which can be resolved at compile time than is immediately apparent.

A procedure in a library package which is not reachable from the main program, can be eliminated from the load. The reuse of common packages is expected to make this fairly common, since not every program will use every subprogram in a reusable package. For example, few programs will use all the subprograms defined in **TEXT_IO**.

3.2.3.1.15 Use of Machine Idioms

These are test problems whose performance can be enhanced by the good use of special properties of target machines. Example idioms include: special instructions to clear memory; store of small constants; reuse of condition code settings to avoid retesting; use of special "loop" instructions; compare between limit instructions; register usage; add-to-memory instructions;

loop instructions which update a counter and test against some limit; memory-to-memory block moves; and memory increment and/or decrement instructions.

3.2.3.1.16 Packed Boolean Array Logical Operators

If an array will fit within a single target computer word the compiler may use fast and small inline code to perform various logical operators, where for larger and/or dynamically determined sizes of packed arrays, a runtime support library routine will usually be called at execution time to perform the logical operation. This can be considered as a type of machine idiom, using the capabilities of the machine.

3.2.3.2 Effect of Pragmas

The LRM defines several pragmas which can affect the performance (time or space) of the generated code. These include CONTROLLED, INLINE, OPTIMIZE, PACK, PRIORITY, SHARED, and SUPPRESS.

The LRM also permits implementations to define additional pragmas, some of which may affect performance. The ACEC is designed to study the performance of Ada implementations as defined by the LRM. It will not contain test problems for implementation defined pragmas. However, individual users are free to modify the test problems to incorporate additional pragmas and observe their effects. They should execute the tests as distributed so that they may compare results between systems; however, they may be interested in exploring the effects of implementation defined pragmas. For a selection process, it would be reasonable to experiment with a few problems to find the setting of pragmas giving the best performance: or even better, to observe the effect of the pragma setting planned for use on the project.

The Association for Computing Machinery, Special Interest Group on Ada, Ada RunTime Environment Working Group (ACM-SIGAda-ARTEWG), has proposed a set of pragmas which might be implemented by multiple vendors. If it is widely supported, a future release of the ACEC might consider developing explicit test problems to reveal the performance impacts of these pragmas.

3.2.3.3 Static Elaboration

It is sometimes possible and profitable to elaborate an object before it is required to do so in a "canonical" order. For example, a library package containing a constant array with static bounds initialized with literal values might be elaborated and initialized before the program is loaded. That array would then not require any overhead when elaborating the package — depending on the other contents of the package, it may not be necessary to execute any code to elaborate the package. In general, if the bounds are not static, it will be necessary to execute

the expressions providing the bounds, at some time (not completely specified by the LRM) after the program is loaded but before any objects in the package are invoked (subprograms called, variables referenced, or types used).

3.2.3.3.1 Aggregates

Frequently arrays (or records) are initialized with an aggregate which can be evaluated at compile time, often with literal values specified. This permits several optimization techniques to be applied.

One approach to aggregates is to consider them as a sequence of individual assignments. However, for a static aggregate, it may be better to define a copy in memory and do a block move into the object to be initialized — block moves are often much faster and shorter than a sequence of load and store instructions. If a static aggregate is used to initialize a **CONSTANT** object (or if the compiler verifies that no assignments are made to the object), an optimizing compiler may make all references to the object refer to the pre-allocated static block, and save both the space for the object on the stack and the time to copy the initial values into it.

3.2.3.3.2 Tasks

Many application designs which use tasking can use a static task structure, creating all tasks when the program is initiated and keeping them active until the program terminates. If an optimizing compilation system, and particularly an RTS, recognizes this, it may be able to save a significant amount of space and time. Space can be saved in the RTS because the internal routines associated with task creating and termination will not be necessary. Time can be saved in two areas. First the time to create the tasks can be saved. This would be done once on program initiation and is not likely to be significant. Second, the RTS routines associated with rendezvous could be replaced with versions which do not test to verify that a task exits. If the number of rendezvous is large and the RTS is organized so that replacement is feasible, the time savings may be significant. In preliminary versions of Ada, there was a provision for static or dynamic tasks.

An optimizing compiler may be able to determine that a task is static either through:

- User declaration, via an implementation defined pragma; or
- By analysis, observing during compilation and linking that all tasks will be elaborated at the library level.

Objects initialized with a static aggregate in the declarative region of a library package will, by definition, only be elaborated one time. This makes it impossible to use the normal ACEC

timing loop to measure the speed of elaborating library units. The special technique used to measure library package elaboration is discussed in Section 3.2.6.5.

3.2.3.4 Language Specific

The definition of Ada tasking provides scope for some optimizations which have no analogue in more traditional languages. This section is entitled Language Specific optimizations, although the techniques described here may be applicable to other languages which support facilities similar to Ada, such as Concurrent C.

3.2.3.4.1 Habermann-Nassi Transformation For Tasking

The Habermann-Nassi transformation for tasking is a technique to reduce the number of task switches required to execute a rendezvous by executing the code of the rendezvous in the stack frame of the calling task, rather than in the frame of the entered task when the entering task is ready to accept a rendezvous when the entry call is made.

Habermann observed that many of the tasks that arise in practical applications are of the "server" type, consisting of one or more *select* statements enclosed in a loop. This structure permits an optimizing compiler to eliminate the rendezvous by replacing the **ACCEPT** statement linkage with a less general but more efficient subroutine which implements the required mutual exclusion and synchronization. A more detailed discussion of the approach is contained in the article "Task Management in Ada — A Critical Evaluation for Real-Time Multiprocessors," by E. Roberts, A. Evans, C. Morgan, and E. Clarke, Software: Practice and Experience, Volume 11, Number 10, October 1981.

3.2.3.4.2 DELAY Statement

Elapsed time measurements of DELAY statements requesting positive delay values will not follow the ACEC product model for analysis (refer to Section 6.2 for details). It would not be desirable if one system which is typically twice as fast as another executed a DELAY statement twice as fast if to do so would require execution time shorter than the requested DELAY value. Comparing execution times of DELAY statements is also complicated by system dependencies because the precision of the requested DELAY value must be expressed in terms of the system dependent type DURATION — a request for a 100 millisecond DELAY can be interpreted very differently by different systems because of truncation and rounding required to convert the value into the proper type.

The DELAY test problems are given a special error code to prevent them from being processed as "normal" test problems by MEDIAN. The measured execution times for test problems executing DELAY statements are printed for examination. Because the actual elapsed time to

complete a DELAY statement can be much larger than the requested value due to quantization and scheduling overheads, this information will be of interest to application programmers.

The measurement of statements with a zero delay value are of particular interest because they can provide insight into system task scheduling and could be given special treatment. These tests are expected to highlight differences between compilation systems.

It is permissible for a compilation system to optimize a literal DELAY 0.0 into a null statement. The Ada Uniformity Rapporteur Group (URG) has recommended that implementations consider a "DELAY 0.0;" statement as a scheduling point — in particular, this would require an implementation check whether a task has been made abnormal (that is, aborted by another task) and terminate it.

Individual ACEC problems are designed to test:

- Does the system treat a "DELAY 0.0;" statement as a null?

Because it is a permissible interpretation, the ACEC should not treat a system which does this as erroneous. Systems which do not treat this as a null could still provide special handling for a delay statement with a literal zero which is faster than when the delay value is not determinable at compile time.

- Is the performance of a "DELAY 0.0;" statement different when there is only one active task in the system than when there are multiple tasks?
- Does the system recognize a "DELAY 0.0;" as a synchronization point for abort statements?

A system which translates this into a null at compile time might "miss" some synchronization points a programmer thought were present in a task. For a task with a DELAY zero in a loop, it could mean that after the task was made abnormal (that is, aborted by another task) the time until it was terminated could be indefinitely postponed.

- Will a zero DELAY force a task switch between equal priority tasks?

By comparing the number of task switches in a series of long running problems, the ACEC can determine:

- Whether the system is using a run-till-blocked or a time-slicing task scheduler. This is tested in problem DELAY_ZERO6X.
- The time quantum on systems which use a time-slicing scheduler. This is the time interval when the task scheduler will switch between equal priority tasks.

Many implementations provide system dependent features which permit a user to specify the task specific schedule algorithm:

- * A pragma, as in DEC Ada

- * A linker directive, as in ALSYS on the Apollo
- * A subprogram which can be called at execution time, as in TLD

The task-switch time can be inferred by running the "same" problem with different task scheduler directives for equal priority tasks, such as run-till-blocked or time-sliced with a short quantum. It will be the difference in measured times divided by the difference in the number of task switches.

3.2.3.4.3 NULL statement

A null or a sequence of null statements might generate some code. There are test problems to explore this.

SS0 is a single null statement.

LABEL_TEST contains a sequence of labeled null statements. It would be interesting to know whether a system generated code for each occurrence of a label.

3.2.4 Performance Under Load

There are some language constructions which display nonuniform performance. The more of them in a program, the slower the average performance. *The simplest example* is program size. Some compilers will not generate as well optimized code for the same arithmetic expression in the last statement of a ten thousand statement procedure as when it is the last statement in a ten statement procedure. Fixed size internal compiler tables for optimization can overflow and the compiler simply stops trying to generate optimized code.

Other examples are in the following sections.

3.2.4.1 Task Loading

The time to perform a rendezvous might degrade as the number of tasks in the system increases.

3.2.4.2 Levels of Nesting

The time needed to access a variable can vary with the lexical level it is declared at. If a "static-link" approach to accessing objects in an intermediate (non-local, non-global) scope is used, the time to reference variables will vary based on the number of links which

must be followed (and on whether registers have been setup by prior statements containing the values of these links). A "display" will provide essentially constant access times, but the overhead to maintain it on block entry/exit can be higher than with a static-link approach. For more discussion on display vs static-link, refer to any of several textbooks on compiler construction: for example, Compilers: Principles, Techniques, and Tools, by A. Aho, R. Sethi, and J. Ullman, Section 7.4, Run-Time Environments, Access to Non-local Names.

The performance of control structures, particularly **FOR** loops, can vary with the level of nesting. Some compilers try to dedicate registers to hold **FOR** loop indexes, and as the level of loop nesting increases, the number of available registers decreases, and the time to load and restore environments increases on subprogram calls.

3.2.4.3 Parameter Variation

Some implementations try to make access to simple scalar formal parameters quick when there are few of them — passing them in registers, for example. Therefore, access to the first (or last) few formal scalar parameters may be significantly faster than access to other parameters. Also, calling subprograms with only a few parameters may be much faster than calling subprograms with many parameters. Input parameters with default values also need to be tested.

3.2.4.4 Declarations

Many target machines have different formats of instructions to be used with different displacements. Code to access a variable which can be reached with a short displacement from a base address can be shorter and faster than code to access a variable requiring a long displacement. When a compiler allocates variables in canonical order, the time to access a variable declared at the beginning of a declarative region might be faster than the time to access a variable declared at the end of the declarative region. An optimizing compiler might allocate variables to memory so that variables frequently referred to are accessible with a short displacement instruction.

Similar effects might be present with respect to access to fields of records.

3.2.5 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. One classical example common to block structured

languages is the tradeoff between a display and a static chain for access to intermediate lexical scoped variables. Here a static chain approach trades off faster subprogram linkages for slower access to intermediate scoped variables. Other examples are discussed in the following paragraphs.

3.2.5.1 Design Issues

There are times when a programmer must make design decisions and the information to determine the best choice for a particular program is not available. Alternately, there are design issues where a programmer may want to know how a system implements various constructions so that a decision to use or avoid the language construction can be made on the basis of quantitative information.

3.2.5.1.1 Order of Evaluation

Order of evaluation is discussed under classical optimizations. See Section 3.2.3.1.12.

3.2.5.1.2 Default vs Initialized Records

Records with default initializations are provided for in the Ada language. If an explicit initialization is specified for all occurrences of the record type, a good compiler would not have any extra code associated with the default initialization. When there is, users may wish to avoid giving defaults.

3.2.5.1.3 Order of Selection

In a **SELECT** statement with several open alternatives, the LRM states that one of them is selected arbitrarily. Several test problems are presented which have several open alternatives to see if some implementations have adapted particularly fast (or slow) algorithms to perform the arbitrary selection.

3.2.5.1.4 Scope of Usage

The relative time to access variables declared at library level, local, and intermediate lexical levels can vary between implementations. For intermediate lexical levels, the use of a display, as discussed in Section 3.2.4.2, can make access times roughly constant; however, this can slow down subprogram linkages. Access to variables declared in the

parent unit of a separate subunit may be slower than what would be observed if the source text were textually included in the parent.

3.2.5.1.5 LOOP Statements

There are alternate ways in which looping constructions can be coded. A **WHILE** loop can be rewritten as an equivalent simple **LOOP** with an **EXIT WHEN** or **IF .. THEN EXIT**, or as **IF .. THEN GOTO ..** statements. There are test problems that explore the performance of these variations.

3.2.5.1.6 CASE Statement

A **CASE** statement with a dense range of alternatives can be simply and efficiently implemented with a jump table on most current machine architectures. When the range between the first and last alternates is large, a jump table approach can result in very large memory usage, perhaps exhausting the addressable memory of the target machine in one statement. Therefore, "sparse" **CASE** statements need to be translated as a sequence of tests. An implementation may choose to implement all **CASE** statements with a sequence of tests. Users will want to know the performance of both dense and sparse **CASE** statements. The ACEC contains examples of each.

3.2.5.1.7 Subtypes

A packed array of a subtype might be stored as an array of the base type, or with a size which permits tighter packing.

3.2.5.1.8 Generics

Generic units can either be shared or treated as templates for macro expansion. When a generic unit is instantiated several times, a macro expansion approach can result in more space than a shared code approach. For macro expansion, depending on the actual generic parameters, it may be possible to reduce the time and space of the generated code (for example, by folding or eliminating unreachable code, which is only unreachable with the particular actual parameters specified).

3.2.5.1.8.1 Text_IO Elaboration Time

The objectives for this set of problems is to examine the performance of tests which elaborate generic packages defined by TEXT_IO. Because of differences in approaches to processing generic instantiations, these test problems are expected to highlight differences between implementations.

TEXT_IO is a predefined library package. It is not generally possible for a user (including the ACEC test suite) to modify TEXT_IO to insert code to force time stamping to be recorded; neither it is feasible to produce multiple versions of the package so that each can be elaborated.

It is feasible to measure the time to elaborate instantiations of the generic packages defined in TEXT_IO: such as FLOAT_IO, INTEGER_IO, or ENUMERATION_IO.

Because they might be treated differently, there are test problems which contain:

- Sharable and non-sharable instantiations.
- Instantiations performed in library units and in nested units.

3.2.5.1.9 Library Subunits

It is possible that a compiler will implement an option to perform INLINE substitution and simplification on subunits. The presence of subunits can interfere with optimizations of the parent unit because the compiler must assume that the subunit may modify any or all objects to which it has visibility.

3.2.5.1.10 Exceptions

One common approach to handling exceptions for faults detected by hardware, such as divide by zero or numeric overflow, will execute instructions on the entry and exit of each frame to track the exception handler which should be called if an exception is raised. Another approach builds a table of ranges so that when a machine fault is detected, the table can be searched to find the appropriate exception handler.

In one case, there will be additional overhead on each frame entry, but comparatively quick selection of the appropriate handler when a fault is raised. In the other, there will be no overhead on frame entry, but slower processing to find the correct handler after a fault is detected. It is expected that raising an exception will be a relatively rare occurrence, compared to frame entry, and that the tradeoff will usually favor the second approach. Users will want to know which approach has been used by an implementation.

3.2.5.2 Context Variation

3.2.5.2.1 Different Coding Styles

The test suite contains test problems constructed to reflect the difference that coding styles (computing the same results through different methods) can make.

3.2.6 Operating System Efficiency

The test suite contains problems addressing those aspects of an Ada RTS which traditionally have been the province of Operating Systems.

3.2.6.1 Tasking

This was discussed under Section 3.2.3.3.2. The simplest way to find the test problems associated with different aspects of tasking is to refer to the VDD Appendix V, "ACEC Keyword Index-1" under task or delay.

3.2.6.2 Exception Handling

This was discussed under Section 3.2.5.1.10. The simplest way to find the test problems associated with exception handling is to refer to the VDD Appendix V, "ACEC Keyword Index-1" under exception handling.

3.2.6.3 I/O

There are several sets of I/O tests.

1. There are tests for asynchronous I/O.

On some systems, any I/O operation will halt the program until the I/O completes. The Ada Uniformity Rapporteur Group has recommended that "A TEXT_IO.GET operation from an interactive input device, such as a keyboard, not be permitted to block other tasks from proceeding while waiting for input. Other types of input/output operations should allow the maximum feasible level of overlap, but it is recognized that in some systems, a general implementation of input/output overlap may be infeasible."

This issue is *not* one of correctness — the LRM does not require, or even discuss, the question of I/O operations blocking other tasks.

Developing a portable test problem to determine this question is complicated because if the system halts waiting for I/O to complete, the test problem may not terminate.

The ACEC contains test problems which measure the performance of operations in one task while another task is waiting on console input. When run on a target which blocks a program whenever a task waits for I/O completion, this test problem will not terminate until the user enters characters on the console. It is possible that having one task waiting for I/O will slow down the execution of other tasks (without necessarily halting them). This is revealed by comparing the results of executing "the same" code with and without another task in the system waiting for I/O.

2. There are tests for console I/O.

Performance of console I/O is not intrinsically correlated with file I/O performance. In conventional operating systems, they will be routed to very different device drivers. The speed of console I/O is important to some applications and needs to be independently tested.

It is not feasible to test console input without using special test equipment, since those tests would essentially measure operator typing time.

These test problems need to be executed interactively — not as a background batch job — because they are intended to measure the performance of console output. The problems are implementation dependent — it is possible that not all target systems will support console output, or that all ACEC users will be interested in console output performance. To work as intended, the target system must interpret an ASCII carriage return control character in a output string as affecting the cursor. The LRM does not require this. These test problems are not necessarily portable.

On many terminals, the time to display a string is a function of the characters to be displayed. The screen management routines of some target systems maintain the current contents of the display and compute the minimal set of terminal commands to modify it into the desired display. For example; a PUT of a string to a line already displaying the same string could result in no physical I/O commands being passed to the terminal. Many terminals have special commands to insert characters (sliding the old contents); delete characters; write blanks from the current cursor position to the end of the line; and overwrite characters. Since the time to transmit a character (either data or control) to a terminal can exceed a millisecond, reducing the number of characters transmitted can be a profitable optimization.

Some systems have bit-mapped terminals where the time to display a line is not correlated to the characters in the line or the old contents of the line.

These test problems will reveal whether the time to display lines varies greatly with the lines being displayed or the prior contents of the line. To enhance repeatability of measurements, the contents of a line will be the same each time the string is displayed. The test problems first write a line of blanks then write two strings in order to observe the time to change between the three display strings. The blanks are written first because on some systems the time to blank a line will be constant and in many of the test problems the first string is the same, permitting users to attribute differences in test problems to changing between the first and second string.

If the console tests only evaluated test problems where the same string was being redisplayed it would give an overly optimistic impression of the performance of systems which omit physical I/O when re-displaying the same string. It would give an overly pessimistic impression of such systems if all the test problems displayed different graphic characters in every position.

3. There are extensive tests for I/O patterns.

Many classes of applications are I/O intensive. Their overall performance is determined primarily by the performance of the file system. The hardware characteristics of the file devices and the efficiency of the target operating system's file processing routines are the primary determinants of the speed of these problems. Projects which develop I/O intensive applications will be concerned with performance of I/O primitives, and will not be particularly interested in knowing how the execution time is partitioned between the Ada runtime system, the operating system, and the physical device.

The LRM, by the FORM parameter on OPEN and CREATE procedures, provides a way for a program to specify implementation dependent options for an external file. A null value is portable and requests the system defaults. On many target systems, specifying non-null values can greatly enhance performance. To ease portability, the ACEC uses null values for the FORM parameter, however, the defaults will not necessarily result in the best performance. Programs concerned with file I/O performance on a particular target are advised to explore the performance differences resulting from specifying non-null values. Examples of such options which may have performance impacts include requests for multiple buffering, read-after-write checking, read-ahead, contiguous allocation on creation, specific devices such as locations for a newly created file, shared vs. exclusive access, etc. Factors of one hundred between default and optimal strings may be common. The default settings for different compilation systems for the same target may not be comparable.

All the following cases consider records of 100 bytes (or variant records with a maximum size of 100 bytes). Resulting file sizes range from 100,000 to 10,000,000

bytes. On many systems, the speed of operations on small files is significantly different from large files, and it is necessary to include some tests problems on large files. By Management Information System (MIS) standards a 10 megabyte file is *not* very large, but it is not trivially small either, and it is not realistic to expect all installations interested in running the ACEC to have much more free disk space than this.

The ACEC contains a set of test problems which produce disk access patterns either typical of important classes of applications or potentially optimizable by common techniques. The set will demonstrate both "typical" behavior and the presence of optimizations.

Typical access patterns include combinations of:

- Sequential

Processing a file in sequential (ascending or descending) order is a common operation in many applications.

- Random

Some processing can be characterized as stochastic, following different distributions.

- * A uniform distribution may be a reasonable approximation to typical access patterns for some applications.
- * Several empirical studies have observed that the frequency of access to records in a file corresponds to the 80-20 rule: 20% of the records account for 80% of the activity, and that smaller partitions of the file follow the 80-20 rule recursively. A pattern of references which correspond to this observation will exercise a system in a typical fashion. The direct application of this pattern corresponds to a hashing on the primary key reference, or if records were accessed through a B-tree, the actual pattern would contain frequent references to the directory pages. Because files are rarely organized precisely in frequency order, a model access pattern based on this distribution should map the frequency based order to a random permutation of the physical order.

The file size makes a major difference in these patterns.

- Cyclic

Many actual reference patterns display a cyclic pattern. For example, an indexed file being used for keyed access will show a typical pattern: Read the root; Read one of the first level directory pages; Read one of the second level directory pages; ... ; Read a leaf page. The number of different pages at each directory level is fixed. The pages in the lower levels are referred to frequently

— for a four level tree, the root page may be referenced every fourth logical I/O operation, and one of the first level directory pages (of which there will be a few dozen different pages) will also be referenced every fourth operation. This pattern can be exploited by locking the root and perhaps the first level directory pages in memory, or more flexibly by using a disk cache.

– Combinations

The above patterns can be combined in different ways. For example, a program may be reading and comparing two sequential files; or a batch update program will process two different files; reading one (the update file) sequentially and making a random access to the second (assuming the master file is hashed).

If disk I/O systems had timing behavior like RAM, it would not be necessary to consider patterns of references. Discs have seek times, rotational latencies, and generally much slower access (compared to RAM). There are software techniques to compensate for the disk performance which are effective for some types of access patterns. Common optimization techniques are:

- Allocating logically contiguous pages of a file to physically contiguous sectors of a disc. This is particularly effective for sequential access patterns because it permits a system to: perform multisector I/Os, transferring a full track at a time minimizing delays for seeking and for latencies; minimize disk head movement when accessing “close by” logical pages of the file; and simplifying and speeding the mapping of logical pages to disk sectors. On some systems, including many UNIX implementations, the file system maintains a directory mapping logical pages of a file to physical disk sectors: and for large files this mapping information is large and forces a (virtual) disk access to the extended file map to refer to logical pages with “large” page numbers. A disk cache will often permit the system to refer to mapping pages without a physical I/O; while this enhances performance, it occupies cache space which would otherwise be available for pages of user files. Other file system designs based on extents can provide a faster mapping between logical file pages and physical disk sectors by allocating blocks of contiguous sectors.
- A disk cache in memory can speed access by replacing physical disk I/O operations with references to the copy in memory. This can be very effective for access patterns with small “working sets.” Examples include directory pages of the file system and high level pages of an indexed file.
- Specifying multiple buffers on sequential disk files, in the same way as is done on tape files, can overlap I/O and processing and reduce the elapsed time required for problem execution. Multiple buffering is applicable independently of contiguous physical allocation — a system can initiate the reading of the

next logical page of a sequential file without this page being the next physical sector. When used in combination with contiguous allocation, both techniques will work better.

- Specifying large blocks on sequential disk files, in the same way as it does on tape files, speeds processing by reducing the number of physical I/O operations required to process a file. Reducing the number of physical I/O operations saves time in the CPU (the device drivers are executed less often) and a multisector read will not insert a complete disk revolution between sectors as might well happen if two consecutive single sector read requests were issued.
- It is well known in the business data processing community that batching together a set of updates and sorting them in the same order as the master file can improve performance. Physical I/O operations might be avoided by processing updates to records on the same disk sector with one operation. Checking the contents of a single buffer is sufficient to insure this. By sorting the updates, the head movement associated with processing the master file can be minimized. Instead of each read referring to a random location in the master file, the sorted list of updates would specify a monotonic increasing sequence with the average distance between specified records being approximately the master file size divided by the size of the update batch. The optimizations appropriate to sequential files, such as multiple buffering, may also be applicable here. For large update files, update processing can be viewed as a merge-like process of reading the update file and the master file and searching for matches.
- The setting of implementation dependent options can impact performance. For example:
 - * Access control. Shared access to a file by concurrent users will involve overheads to insure consistent usage. When a particular file is open reserving exclusive use, processing should be faster because the file system will not need to manipulate record locks.
 - * Allocating files which will be accessed concurrently to different physical drives and/or onto different disk controllers can minimize head movement and channel contention. Not allocating high frequency access data files to the same device where the operating system files are loaded can also reduce contention. Although the best performance for a program may result if every file is allocated to a different physical disc, many installations will not have enough different devices to do this, or may determine that the best installation wide performance will be achieved when every program does not allocate files on every disk (file backup is greatly simplified if one project's files are allocated to one device).

- * Journaling is the recording of file activity to keep audit trails. It is necessary in some applications that such trails be maintained, and some operating systems may provide for automatic journaling — but it does have a performance cost which can be large.
- * Read-after-write checking is often a user selectable option, permitting the tradeoff between performance and the surety resulting from verifying that a disk write was correctly completed.
- * Some systems provide an option to assign a file to memory. When available, this can eliminate all physical I/O operations except for initial load and final save — although if loaded into a virtual memory system, there may be paging I/O operations associated with the "RAM FILE" depending on the available physical memory and the load on the system, but the overhead for this I/O is (hopefully) smaller than for disk resident files.
- * There may be an option to selectively enable or disable a "write-through" option on a disk cache. When enabled, a physical write would be performed on the disk (and the cache) and the users would be assured that the disk file had been updated in the event of a system crash.

FORM strings are implementation dependent. Programs using default FORM strings will be portable, but are dependent on the implementation (and perhaps the installation and/or the characteristics of the account the program is executed under). Two different compilers which generate code runnable under the same operating system may select different defaults. It is important that users understand that specifying explicit FORM strings on some systems can have very large performance effects (orders of magnitude difference). A project which is seriously concerned with file system performance might establish coding conventions which require all programs to specify FORM strings and may be completely uninterested in the performance obtained by using default values. These test problems will construct and use several test files. These will be instantiations of packages SEQUENTIAL_IO and DIRECT_IO for a 100 byte record type. Several sizes of files will be used, ranging from 100 records (a 10 KB file) to 100,000 records. The test program will generate these files during its initialization. The size will be an adaptation parameter so that interested users may explore the performance of different sized files. For many small configurations finding this much free space will be awkward. Relatively few configurations have ten megabytes of buffer space or disk cache, so the test problems should measure disk performance. Organizations developing applications for large systems should explore performance on files as large as they plan to use.

3.2.6.4 Memory Management

There are performance tests which focus on memory management issues. Some test the objects explicitly allocated and deallocated by a program via NEW and the instantiation of UNCHECKED_DEALLOCATION. There is a large set of tests which focus on the management of implicitly allocated objects which are discussed in more detail in the following subsection.

3.2.6.4.1 Implicit Storage Reclamation

Many systems create temporary objects at execution time for Ada statements which do not explicitly contain an allocator. For example, functions returning an unconstrained type will typically allocate space on the heap to contain the function value. There are two issues that follow from this observation:

- First:
Because the performance of storage allocators can vary greatly between systems, test problems which invoke allocators can vary greatly between systems.
- Second:
If the memory space for these objects is *not reclaimed for later use*, available memory will “evaporate” as the program runs, and eventually space will be exhausted and the program will crash. This is a particularly nasty problem because:
 - * The program source *looks* correct, and may work flawlessly on other systems, including earlier releases of the same compilation system. This is a special problem for software that was developed for reusability and portability.
 - * The existence of a problem may not be discovered until late in a project life cycle — perhaps only after the system is made operational. On target systems with gigabytes of virtual memory, it may take a long time to exhaust available memory — not so long that it will not eventually crash, but long enough that the application can pass most testing.
 - * After a crash, it may not be easy to determine the reason for failure. If users have suppressed checking, very strange behavior may result when the system eventually exhausts free space.

An implementation which can execute an Ada statement once but fails when it is executed repeatedly is not very robust.

There are ACEC test problems which:

- Detect if a system allocates and does not reclaim storage. In these cases, error reports against the system should be generated.
- Demonstrate the efficiency of a system in executing statements which (might) implicitly allocate and should deallocate storage. Some systems may be able to perform some of the test problems without allocating from temporary memory. That does not invalidate the test problems because it is desirable to have test problems which demonstrate performance differences between systems.

Because of differences between systems in their efficiency in manipulating temporary memory objects, and the possibility that some systems will be able to support some of the test problems without requiring dynamically allocated temporary memory, these test problems are expected to highlight differences between implementations.

Test problems use a simple strategy to test for storage reclamation. Each suspected language feature is executed 100,000 times in a test problem. If each execution allocates and does not reclaim as much as a few bytes, the space usage will quickly grow and exhaust space on systems without a large amount of usable free space. This will be effective at finding faults on the non-virtual memory systems typical of embedded applications. A size testing technique based on executing programs in a task with a length clause specified might be able to detect faults with fewer problem iterations; however many systems do not support length clauses, and even on those that do, a system might allocate space for implicit objects off the stack where it would not be affected by length clauses. If the execution of these test problems exhausts space and raises `STORAGE_ERROR`, then the test problem will process the exception and report the fact (and not give an execution time measurement).

The elapsed time to execute 100,000 Ada statements for some of these tests could be excessive (that is, many hours to run one problem). The problems make a preliminary estimate of the execution time for 100,000 statements and do not attempt it when it is excessive.

Any other ACEC test problem might have the side effect of exhausting space when it is executed repetitively inside the timing loop, if it contains a language feature for which the compilation system allocates and does not reclaim space.

3.2.6.5 Elaboration

Elaboration in Ada occurs in several contexts which are significantly different with respect to performance, although not with respect to syntax.

For subprograms and blocks, entry into the declarative region implicitly invokes the elaboration of their declarative regions. This is extensively tested for.

The test suite contains problems which elaborate nested (that is, non-library) packages and which perform similar sequences of statements to that which would be performed by a library package elaboration (that is, calling on explicit allocators to get dynamically determined space).

There are several problems which use a modified version of the timing loop to measure library package elaboration time. These versions declare multiple (25) library packages with an elaboration order defined by use clauses and PRAGMA ELABORATE which forces a linear order of elaboration. Using this order, it is possible to place code in the initialization block of the package bodies to perform timing measurements. Although the error bounds obtainable in this way are not nearly as tight as obtainable from the normal timing loop, they can inform the reader of the order of magnitude of the time necessary to elaborate library packages. The elaboration is done so that it is possible to perform some consistency checks.

3.2.6.6 Runtime Checks

There are test problems to measure the cost of verifying that the predefined constraints are satisfied.

There are some problems which contain the same source text where the only difference between the problems is the presence (or absence) of suppression pragmas. Comparison of these problems would reveal possible performance savings by specifying suppression of checking.

There are other sets of test problems designed to test specific aspects of constraint checking code.

3.2.6.6.1 Elaboration Checks

There are some test problems to observe performance where code has several calls on subprograms defined in an external package where an optimizing compiler would be able to combine some of the checking code which verifies that the package body has been elaborated before calls on subprograms defined in the package are performed. Because of differences in approaches to constraint checking and control flow analysis, these test problems are expected to highlight differences between implementations. These test problems must be compiled without suppressing checking for predefined constraints because they are intended to test the quality of the code generated to perform the checks.

A system must verify that a package body has been elaborated before it is proper to call on a subprogram defined in that package. By using the predefined pragma `SUPPRESS ELABORATION_CHECK ( package_name )` it is possible to avoid all checking code when the system honors the request. However, when no suppression pragma (or comparable compiler option) is specified, an optimizing compiler still has available more efficient options than generating testing code before *every* call on a subprogram defined in an external package. For example, it might apply some data flow analysis and only generate one test for pre-elaboration of a package in a region independent of how many different calls on subprograms in the package are contained in the region.

3.2.7 Application Profile Tests

The test suite contains test problems representative of how Ada is being used in practice. In most programs, a small fraction of source text accounts for a large fraction of the execution time of the program. The test suite contains examples of such time critical sections of code extracted from MCCR applications. The performance of an Ada compiler on these examples will be a good estimator of the expected performance on a similar program. These examples will be selected to represent typical Ada usage. Neither code complexity, nor use of specific language features are selection criteria. They can result in test problems which look like "FORTRAN with semicolons," but if that is the way the language is being used in practice, then the ACEC will contain test problems representative of this usage.

The following subsections discuss, in turn, classical benchmarks, Ada in practice, and "ideal" Ada.

3.2.7.1 Classical Benchmark Programs

The test suite contains classical benchmark programs coded in Ada. Examples include: Whetstone, Dhrystone, Livermore Loops, Ackermann's function, GAMM, sieve, puzzle, several sort routines, the eight queens problem, problems from the Computer Family Architecture study (LU, BMT, TARGET, HEAPIFY, and AUTO), and Ada versions of the inner loops discussed in the paper by D. Knuth "An Empirical Study of FORTRAN Programs" in Software: Practice and Experience, Volume 1, Number 2, 1971.

3.2.7.2 Ada in Practice

There are example test problems drawn from Ada programs extracted from projects. They represent typical usage of Ada. Examples are drawn from:

- A simulator for the E-3A. This is a set of problems drawn from a flight simulator. There are eight test problems from navigation, avionics, and communications. These are in program SIMULATE.
- The Advanced Rotocraft Technology Insertion program, from Navigation and In-flight Performance Monitoring modules. These are in the program ARTI.
- Radar tracking algorithms. These are in program SA8TEST.
- A set of test problems which manipulate a balanced tree (insert, delete, search) are provided in program AVL.
- A set of test problems which manipulate a trie (insert, delete, search) are provided in program TRIE.
- A set of test problems which perform an A* search over a graph are provided in program A_STAR.
- A set of test problems which exercise a neural network are provided in program NEURAL.
- A set of test problems which perform Cyclic Redundancy Check (CRC) operations are provided in the program CRC. This is an Ada implementation of the common application in communication programs.

- State information

The passing of program state information to a routine can be performed in several ways. This is a frequent operation when an object-oriented coding style has been used. It can be useful to inform users of the performance impact of different styles. One specific example has been selected (a lag filter) and is coded in different ways, consistent with object orientation. The significant points about the variations are the way state information and input/output are handled.

- * One style uses procedure parameters, where the calling procedure would retain the state information.
- * Another alternative is to code it as a generic unit and include the state information as part of the generic parameters. That is, each instantiation would retain its own state and a procedure to advance the state of the system one time interval would not have any explicit parameters because the identification of the input and output variables, the filter coefficient, and the state variables would all be encoded into the generic unit.
- * Another approach is to have a generic unit where the instantiated procedure to advance time would retain explicit parameters for input and output and perhaps state. These coding styles represent different design approaches and users may

be interested to know if there are significant performance differences associated with the different approaches. Inlining could be specified for either approach.

- * Another natural approach is to define a library package containing the state variables and the access procedures for each filter object in the design. This involves duplicating source code for each filter object being modeled.

3.2.7.3 Ideal Ada

Ada in practice may be criticized because current practices represent techniques adapted from experience in other languages, and may be just "FORTRAN with semicolons." Such programs may ignore Ada language features not present in prior languages, such as exception processing, or tasking. The ACEC contains problems initially designed with Ada capabilities in mind, which do not artificially constrain themselves to a subset of the language. Programs designed for FORTRAN or other languages and coded in Ada will probably not use Ada features not present in the design language, such as tasking, exceptions, packages, user defined types if originally designed for FORTRAN, etc. The Artificial Intelligence application problems were originally coded in LISP, and the code is not restricted to FORTRAN or JOVIAL constructions.

3.3 CODE SIZE EFFICIENCY

The memory size of programs is an important attribute to many MCCR 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).

3.3.1 Code Expansion Size

The code expansion size is measured in the timing loop (see Section 5.5). It is the space, in bits, between the beginning and end of each test problem. This is an important metric to many users.

3.3.2 Runtime System Size

The size of the RunTime System (RTS) 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.

Refer to the Section 6.5 on RTS Size in Statistical Background for MEDIAN for further discussion.

3.4 COMPILE TIME EFFICIENCY

The times to compile the compilation units are collected and analyzed. The programs 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.

The compilation time is the time to translate from source text into executable modules. This definition includes time spent in linking intermediate objects into an executable file. To exclude time spent in linking would not give a realistic measure of the actual time required to compile programs.

The ACEC measures the time to compile and link source programs. The analysis in MEDIAN does not worry about how to count lines to compute a lines-per-minute compilation rate. However, the SSA program does compute lines-per-minute.

There are several features which may impact compile speeds:

- **Size.** Several code generator paradigms build a representation of a program (or a piece of a program such as a subprogram, basic block, linear sequence of statements, ...) and perform various manipulations on the structure to try to produce good code. Some of the algorithms used are of more than linear time complexity — they run much slower on large units than on short ones. Hopefully, in exchange for taking more time for compiling, better code will be generated.
The time associated for each phase of a compiler may mean that all programs take some fixed time. A short program may not take much less time than a slightly longer one.
- **Generic instantiation.** The amount of time associated with instantiating a generic unit can be substantial. The compiler must check that type signatures match, and if it is treating instantiations as a form of "macro expansion" it may try to optimize

generated code. When an actual generic parameter is a literal, the compiler may use this value to fold expressions in the generic body.

- Each library unit referenced in a **WITH** clause can require considerable processing time, including the time required to search the program library for the definition. The state of the program library might greatly influence the compile time performance. Searching a program library which contains several hundred units can be much slower than searching a library which is nearly empty. Some implementations may use a serial search or may hash their search. Allocated space for new objects in the library may be dependent on the status of the disk space. If the available disk space is fragmented, finding a large block of contiguous storage may be time consuming, and linking together noncontiguous storage may slow down all later accesses to the library object.

Note that error messages and user friendliness (creation of cross reference listings, set/use lists, text formatting, ...) are important aspects of compilers which can influence speed of compilation but whose utility may well be worth a serious degradation in raw speed.

The definition of obsolete units in Ada will require the recompilation of some units when some changes are made to other units. Several Ada compilation systems provide user support to identify dependent units, and in some cases to automatically recompile all effected units to bring a program library "up-to-date." The presence of such facilities can be helpful and productive; however, it needs to be said that some recompilation orders, although valid, will result in recompiling the same unit more than once. This is an obvious performance problem, and if a system supports an automatic recompilation facility, users will want to know how efficient it is. Such facilities are system dependent and are not addressed in the ACEC performance tests. Automatic recompilation and other facilities provided by the Ada program library management system are tested in the ACEC Library Assessor.

3.5 TESTS FOR EXISTENCE OF LANGUAGE FEATURES

The emphasis of the ACEC is on performance. Testing correctness of implementations is the charter of the Ada Compiler Validation Capability. (ACVC).

Some test problems are constructed to measure the performance of language features which may not be supported on all target systems. If these test problems fail, then a user will know that the target system does not support the feature. Test problems for some implementation dependent language features may need to be adapted for individual targets. This includes tests for tying tasks to interrupts.

3.6 USABILITY

Implicitly, by compiling and executing a large set of Ada code, a user will get some idea of the usability of a system. In compiling and running any substantial volume of Ada code through a system, users will learn something about how that system works and how convenient it is to use. Although this is not a quantitative assessment, it should not be dismissed lightly.

There are explicit tests for the usability of several aspects of a system, including symbolic debuggers, Ada program library managers, and diagnostics.

3.6.1 Symbolic Debugger Assessor

The results of executing the ACEC debugger assessor scenarios will be a template which the user has completed to reflect their findings. In reviewing the template, there are several points which a reader must keep in mind, and these are detailed in this section.

Some systems may provide a machine-level debugger which is ignorant of Ada symbolic names and linguistic constructions. On such a system, to display the value of an Ada variable, the programmer would map the Ada variable name to a memory address and examine that address — the compiler/linker may print this mapping information in listing files. As a general rule, such machine-level debuggers are not as convenient to use as symbolic debuggers, although they typically impose few restrictions on the programs to be executed under the debugger. The ACEC debugger evaluator is not intended to evaluate such a tool.

The LRM does not levy any requirement that an Ada compilation system provide a symbolic debugger, nor any standard set of capabilities or operational interface which it should use if one is provided. The ACEC debugger scenarios are designed to emphasize the determination of functional capabilities — they specifically do not consider the elegance or efficiency of the user interface, although this may be very important to users in addition to the functional capabilities of a debugger. Some organizations may want to evaluate the efficiency of a debugger interface by measuring elapsed time or by counting either keystrokes or commands. They must be careful to insure that similar approaches are used on the different systems. Measuring elapsed time may result in evaluating the typing speed of the operator more than any inherent properties of the debugger or target system, and may be particularly misleading when a programmer has to stop and read a manual for an unfamiliar debugger in the process of performing the scenarios. Counting keystrokes or commands can also be misleading: a system which permits macros or user defined function keys can arrange to execute an entire scenario with one keystroke. This

may be very non-representative of typical performance — although users could define a macro that would save more than one keystroke every time, very few users actually will do this unless they anticipate using the macro multiple times. Organizations evaluating several systems must decide what “comparable” usage on the target systems would be. This will generally require a case-by-case comparison of each system’s facilities and the macro/function-key usage the organization anticipates will be typical in their projects.

The functional capabilities to be tested were selected after a review of the capabilities of existing debuggers (for Ada and other languages) and capabilities whose lack has hampered the debugging of programs in previous systems. No priorities were assigned to the individual capabilities; each organization may have its own priority ranking of debugger capabilities. The template separately lists each capability so that users can easily see how the systems differ. All the scenarios are not of equal importance. For example, a debugger which can perform all the scenarios except for not being able to examine the value of any variable would probably be of little practical value.

The ACEC debugger template provides a place for users to report subjective assessments and general comments. While these are not easily compared between different systems, or between different individuals using the same system, the information they provide can be valuable.

The ACEC debugger scenarios are designed to ask specific questions in the context of specific programs; the results should not depend strongly on the experience of the evaluator. However, the completed ACEC debugger template will reflect judgments made by the evaluator. Running the ACEC debugger scenarios may be the first time a programmer actually uses a debugger. The reports may reflect the system documentation more than the debugger.

A user should remember that the inability to perform a scenario may be due to faults in the debugger documentation instead of the capabilities of the debugger. While the system might be able to perform the scenario, the documentation may not explain the debugger well enough for the evaluator to discover a way to do it. If the documentation is unclear, the user may call on the vendor for support. Even with adequate documentation, determining whether a particular scenario is possible can involve a subjective judgment on the part of the evaluator. If the debugger can single-step through a program and if the programmer is very patient, almost any scenario could be performed. For example, on a debugger which does not support watchpoints on variables, a programmer *could* stop after every statement and examine the variables to be monitored.

3.6.2 Program Library System Assessor

The library assessor template will be filled out to reflect the capabilities discovered in running the library scenarios. The primary purpose of the program library system assessor is to determine the functional capabilities of a system, although it also collects some performance data (elapsed time and disk space size) and will determine whether the capacity of a system is large enough to accommodate the provided scenarios.

The LRM levies only minimal requirements on a program library system — after a unit is compiled into a library it shall be possible to subsequently compile units which reference it. The LRM suggests (Section 10.4 paragraph 4) that a programming environment provide commands for creating the program library of a given program or of a given family of programs and commands for interrogating the status of the units of a program library. The form of these commands is not specified by the LRM.

Ada compilation systems provide program library systems with different design approaches, functional capabilities, and efficiencies. The ACEC provides information for users to assess the functional capabilities of systems (and to a lesser extent, their efficiency). The different design approaches determine the framework in which the operations are performed — they are not directly evaluated. The ACEC library assessor approaches are based on providing a set of scenarios consisting of compilation units, operations to perform using them, and instructions for evaluating the system responses.

An ACEC user will have to adapt each scenario to the target system. On reviewing the completed template, an ACEC report reader must be aware that a statement that a capability was not found to be present does not necessarily reflect a failure in the system — the tester could have overlooked a supported capability, or the execution of the scenario which would determine the capability might have exceeded the capabilities of the configuration. For example, there might not have been enough free disk space on the test system to enter all the compilation units of a scenario into a program library, which is a very different result than a system which would not accommodate the scenario if sufficient resources were available.

Different organizations will assign different priorities to different capabilities. Support for concurrent library access will be critical to projects which will have cooperating multi-programmer teams; it may be unimportant to single-user standalone systems.

The ACEC scenarios emphasize determining functional capabilities. They do not attempt to evaluate the elegance or efficiency of the user interface — the scenarios are equally applicable to a "point and shoot" graphic based user interface and to a command-line based user interface. They do not try to count the keystrokes or mouse clicks necessary to perform an operation (for any system supporting a macro capability this would be

awkward; an evaluator wanting to make a system look good would define an entire scenario as a one character command). Comparing the sequences of keystrokes (or mouse-clicks) required to perform the different scenarios on different systems can be interesting. Comparison of the command sequences from different systems will provide readers insight into the operations of the systems.

Library systems provide a structure in which Ada programs for the compilation system will be developed. *Until a user understands the structure that a library management system is designed to provide*, the user will judge all operations in it to be awkward. They may consider all operations to be awkward even after they understand the compilation system's library design as well, but such an assessment is properly made *after* initially learning the system. The ACEC scenarios are designed assuming a "typical" library system design, which has mapped fairly easily to the sample systems tested during the development of the ACEC — but that is no guarantee that it will map easily to all implementations.

A library system may have features not exercised by the set of scenarios. The library assessor template provides a space for user comments which can be used to report additional capabilities. Readers should review any comments in this area and decide how important they consider the additional capabilities.

3.6.3 Diagnostic Assessor

The ACEC diagnostic assessor template and report will be completed to reflect the discoveries made in running the diagnostic tests. In reviewing the template and report, there are several points which a reader must keep in mind, and these are detailed in this section.

The LRM requires that a compilation system reject illegal programs, but it neither specifies the form/contents of diagnostic messages, nor exhaustively lists the conditions which should generate warning messages. The ACEC diagnostic assessor tests include examples of illegal programs where the intent is to determine whether specific points are mentioned in the diagnostic message which would help explain and isolate the problem. The ACEC diagnostic assessor tests also include examples of programs where a helpful compilation system would generate a warning message stating that the code, while not *illegal*, contains "suspicious" constructions and may contain logic errors or inefficiencies. On reviewing the completed template and diagnostic summary report, an ACEC reader must be aware that the user has made a judgment about whether the generated message contained the anticipated information.

Each organization may have its own priority ranking of classes of diagnostics. The ACEC template and report separately present the categories so that users can easily see how

the systems differ. Validated systems will reject illegal programs, more or less clearly. Large differences between systems occur with respect to the processing of warnings, and to a lesser extent in the presentation of non-local information.

The ACEC diagnostic template provides a place for users to report subjective assessments and general comments. While these are not easily compared between different systems, or between different individuals using the same system, the information they provide can be valuable.

The number of diagnostic assessor tests is fairly small. There may be some compilation systems with generally good diagnostic messages which happen to do poorly on the particular examples included in the ACEC diagnostic assessor tests; or a system with generally poor diagnostics may do well on the ACEC examples. If implementors start to "tune" their systems to do well on the ACEC diagnostic assessor examples, the ACEC results may not reflect a "good" sample of test cases. This is a larger risk for the diagnostic tests than for the performance tests because the relatively small number of examples makes it easier to modify the ACEC results by "small" changes in the compilation system.

3.7 CAPACITY TESTS

There are no test problems which test capacity limits in the sense that a problem is run to see how many objects can be allocated in a collection of a specific size, or how many tasks can be created on a system. All problems in the ACEC are designed to be portable tests of performance. A test problem to determine the maximum number of objects which can be allocated, explicitly or implicitly, until memory is exhausted will execute a different number of Ada statements on different systems, depending on what the capacity limits on the target are. The time to run such a test depends more on what the capacity limit is than on the "average" performance of the system. As such it would not be a well formed ACEC test problem.

A program exceeding a capacity limit can sometimes be adapted so that it will run. For example, a 'STORAGE SIZE clause could increase the size of a task's local storage, or a linker directive could specify the size of the global heap. Such tuning often requires a good deal of user experimentation and time to perform. The effort is not necessarily easy, although some systems may give helpful advice to assist the process. For users not experienced with the target system, "tuning" the space parameters will be a learning experience. The learning time to understand a target well enough to tune it may exceed the effort an organization is willing to dedicate to an ACEC evaluation. Readers must understand that in such capacity limited problems, the fact that a tester was not able to

get the problem to execute may simply reflect the experience and effort they allocated to the problem — it may be possible the problem would fit if sufficiently “tuned.”

Some problems may fail because they exceed some capacity limits in the compiler or on the target machine.

3.8 SUMMARY

It should be clear from the above discussions that the ACEC is not a simple collection of problems with a hierarchical structure.

It may be simplest to view the test suite as accessible from several different perspectives, any of which may be of interest at different times. The indexes in the VDD (Appendix V, “ACEC KEYWORD INDEX-1” and Appendix VI, “ACEC KEYWORD INDEX-2”) are designed to simplify access, depending on areas of interest.

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 either unusually well (or unusually poor). These problems can then be examined in more detail to try to determine what characteristics of the problems were responsible for the unusual behavior. Of course, measures of overall performance are also collected and will be useful in comparing systems.

3.9 SIMPLE STATEMENT TESTS

There are many SIMPLE STATEMENT test problems, named SSnnnn where “nnnn” varies up from 0 to the maximum number of SIMPLE test problems. These problems typically test a fairly isolated language feature or syntactic construction.

There are many test programs which contain multiple test problems. This is desirable to minimize the time required to execute the programs on embedded targets where downloading can be a time consuming operation. This packaging decision introduces a potential difficulty for a user wanting to match up a test problem with the program in which it is contained.

The number of individual test problems in any one test program will be limited to reduce the risk of exceeding capacity limits and to reduce overheads associated with manipulating many separate, small programs.

The assignment of simple statement test problems to programs is not entirely arbitrary. Some problems use locally defined objects, and care must be taken to insure that the intended set of suppression pragmas apply to the test problems. Many compilation

systems are not fully supporting the suppression pragmas as defined in the LRM. Some are requiring compilation unit wide scopes for the pragmas, which would not make it possible to include in the same program a test problem with and without suppression being specified.

There is a simple program naming convention for simple statement tests which will permit users to easily correlate problems and programs and not impose packaging problems. Define the names of the simple statement programs as follows:

SmmmmTnn

Here the field "mmmm" is the number of the first simple statement test problem in the program, and "nn" will be the number of the last simple statement test in the program (mod 100). For example, the first compilation unit is named "S0000T14". This system produces names with eight characters and follows the same format even if only 1 simple statement test is in the program (there are no programs named SSnnnn). The convention is adequate for up to 10,000 simple statement test problems, which provides a sufficient margin for further development before running out of name space.

The numbering of Simple Statement test problems reflects the order in which they were developed. The numbering does not reflect the importance of the problems — problem SS256 is not necessarily more, or less, important than problems with lower numbers.

4 HOW TO INTERPRET THE OUTPUT OF THE ACEC

There are three primary sets of output available to the reader: the OPERATIONAL SOFTWARE results file generated by executing the benchmark test suite, the automated analysis produced by the multi-system comparative tool MEDIAN, and the SINGLE SYSTEM ANALYSIS (SSA) report. These will be discussed in turn.

4.1 OPERATIONAL SOFTWARE OUTPUT

Each test problem, when executed, generates a standardized output describing the timing and code expansion size measurements produced when the test problem is executed.

There is external information which an ACEC reader should know about a system to properly interpret the significance of a set of results.

- Identification of the system being tested

- * Version number of compiler and the compiler options used for the different command files.
- * The operating system version number and relevant tuning parameters (*priority*, available memory, etc.).
- * Hardware characteristics: amount of memory, type and number of disk drives (which can greatly impact the compile speeds).
- * Whether other users were on the system when measurements were made.
- * The dates the measurements were made.

- Identification of tuning performed

In particular, the reader must know what math library was used:

- * an implementor supplied generic math package conforming to the Association for Computing Machinery, Special Interest Group on Ada, Numerics Working Group (NUMWG) - this is the recommended one;
- * or an interface to an implementor supplied library;
- * or the ACEC supplied generic math package using the implementation independent version of MATH DEPENDENT;
- * or the ACEC generic math package using a version of MATH-DEPENDENT tailored to the target.

For a detailed discussion of the alternative ways of adapting math, refer to the ACEC User's Guide, Section "ALTERNATIVE METHODS FOR MATH".

When comparing two different compilation systems, it is possible that performance differences in the test problems which used math functions may be due to the technique used to implement the math library. An optimized, vendor supplied math library might be considerably faster than the ACEC math library with the representation independent version of MATH.DEPENDENT. Because the NUMWG recommendations have not been included in the Ada language standard, and because no implementation is required to provide a math library conforming to NUMWG (or to provide any math library at all), the ACEC includes a math library designed for maximum portability. If a major project is committed to using a compilation system, it might invest the resources to develop (or adapt) an optimized math library which is tuned to the target and which provides a significantly better performance than the ACEC portable math library.

4.1.1 Results File

This section describes the format of the results file produced by the execution of the test suite and the significance of the fields and values printed. Consider the following figure (column indicators are added to aid the reader in locating column positions).

The output is a sequence of fields, sometimes spread over several lines. Numeric data is right-justified within its field with leading zeros replaced with blanks.

1. The name field identifies the test problem. This field begins in column one of the results file and is terminated by the first blank, horizontal tab, or new line character. The name of a test problem is constrained to be unique within the test suite. The first column of subsequent lines will be blank.
2. The descriptive field which follows the name field provides additional information about the test problem. For short test problems, this field can contain the complete Ada source text. In other cases the field will be a narrative which describes the problem. The test problem description can refer the reader to related test problems, specified after the "cf." The field can span several lines.
3. The code expansion size field is a four column field beginning in column 40 containing the number of bits generated inline for the test problem. How this and remaining fields on the line are computed will be discussed in depth in the next section. Notice that leading zeros become blanks and the number is right justified within the field. It is of the form "nnnn."
4. The minimum time field is a nine column floating point number beginning in column 45 which contains the minimum number of microseconds the test problem took to execute. It is in the form "nnnnnnn.d." Leading zeros become blanks and the number is right justified within the field. This is the time extracted by the FORMAT tool for analysis. For small values of measured time (test problems which execute in less than 10.0 microseconds) the display format uses an exponential output, where a fixed format could truncate results, displaying less precision than the accuracy of the measurements warrant. If this were not done, a change of one unit in the least significant place for a test problem which executes in one microsecond would represent a ten percent change, which is significantly greater than the confidence interval. For large values (greater than 100 seconds) the measurement is also displayed in exponential format so that the field will not overflow and displace the column alignment.
5. The mean time field is a nine column floating number beginning in column 56 which contains the mean number of microseconds the test problem took to execute. It is also of the form "nnnnnnn.d" and will have small and large values displayed in exponential format.
6. The inner timing loop count field is a two column field indicating the integer number of iterations the test problem was performed. The actual number of iterations is 2 ** (value_of_this_field) - 1.

7. The outer timing loop count field is a two column field of the number of times the test problem was repeated using the iteration count displayed in the prior field.
8. The sigma field is the percentage standard deviation observed in the measurements. This field is blank when the mean is zero.
9. The indicator field, which immediately follows the sigma field, is either blank or "#." It is blank when the desired confidence level was achieved. A Student's t-test is used for this determination. This test is discussed in most statistics textbooks, including Introduction To the Theory of Statistics by A. Mood and F. Graybill, McGraw-Hill, 1963.

The fields reporting the timing loop counts and the standard deviation of the measurements are concerned with the reliability of the measurements. Basically, they give a user a feel for the statistical significance of the numbers reported as measurements, and whether they can be trusted. Their computation is discussed in Section 5.

4.2 MEDIAN OUTPUT

MEDIAN analyzes the collected sets of measurement data, as produced by executing the test suite. It computes factors representing both the performance of each test problem and each system.

The background for the MEDIAN analysis is discussed in Section 6.

It is possible for ACEC users to run a subset of the test problems on one system. The problems not executed will be flagged as not attempted and the analysis tools can proceed comparing the problems which were executed on that system with other systems which have data available. While it would then be possible for an ACEC user to run only the test problems addressing the language features of particular interest, this is not the recommended approach for users considering selecting a compiler.

Any complex program will contain some surprises, and an Ada compiler is complex. A decision to run only the problems associated with, for example, tasking, is an implicit assumption that the processing of other language features will be comparable between the other candidate systems, and are of no particular concern.

Each section of the MEDIAN output is discussed in turn in the order it is produced by the MEDIAN program. The sample output used is for a small set of problems (4) on a small number of systems (4).

4.2.1 Raw Data

The raw data is a matrix which contains the actual measurements. As the descriptor page which precedes the matrix says, each column is representative of a system; each row is representative of a test problem. Missing data processed by FORMAT will be indicated with a mnemonic such as "UNRELIABLE," "CMP_TIME," "NO_DATA," etc. These mnemonics and their meanings are discussed in more detail in the User's Guide, Section "PREPARING THE DATA".

In the sample presented, there is nothing exceptional about the raw data.

4 problems handled by MEDIAN.

7 JAN 1988 16:19:18

This is a table listing the measurements. Timing measurements are in microseconds. Space measurements are in number of bits. There is a column for each system and a row for each problem. The table of measurements starts on the top of the next page so that the page alignment between the measured observations and the residual table will be the same.

7 JAN 1988 16:19:18

TEST PROBLEM NAME	VAX_P_VMS	INTEL8086_P	I432_ADA_3	NS16000_P_10
SEARCH	0.34	1.99	2.78	0.47
SIEVE	5.56	6.64	8.07	5.38
PUZZLE	2.44	3.78	5.19	2.48
ACKER	2.29	2.41	5.56	1.25

4.2.2 Histograms

The purpose of a histogram is to graphically display the distribution of residual values. Large spreads in a histogram indicate a system which has a very different performance pattern from the other systems compared. One histogram is constructed for the entire residual data set and one histogram is constructed for each individual system.

The sample data follows on the next nine pages. The first page of the output produced by MEDIAN is a descriptor page containing pertinent information about the histograms to aid the user in interpreting them. With only 4 problems, the histograms are not particularly informative. When there are more test problems, users should check for patterns of residuals. If the pattern is not approximately "normal" (a bell-shaped curve) for a large sample size, the estimates of overall performance factors may not be extrapolated to a variety of test problems, because there are large relative differences in one system's treatment of some language feature (as reflected in the test problems) to another system. For example, consider comparing two systems, one on a target with software simulated floating-point arithmetic and the other the identical machine with a hardware floating-point support option installed. Considering only test problems using floating-point arithmetic, the second system might be a hundred times faster than the first; but this would *not* be a good comparison for test problems which do not use floating-point operations. In this example, proper comparison would depend on the amount of floating-point computations in the applications the ACEC users are interested in.

Because each histogram displays residual factors they will always be centered around the factor 1.0. The overall speed of a system is reflected in its system factor. The histograms displaying the distribution of system performance are useful for two purposes:

- It is a verification of the statistical model used by MEDIAN.

An ACEC user comparing systems which do not fit the MEDIAN modeling assumptions should be cautious in interpreting MEDIAN results.

- It permits an ACEC user a simple way to determine whether a system looks "normal."

A system with many problems with large residuals is performing some problems much slower than the systems it is being compared against, relative to its performance on all other problems. It may be important to investigate these problems to see if they share some common characteristic which can explain the behavior. The ACEC reader must then decide whether the characteristic is important to the application area in which they are interested.

There are several observations to make about the histograms.

- The total number of entries displayed in the histogram is the number of test problems where valid measurements were observed — it excludes errors, problems measured as zero (optimized into null statements), and unreliable measurements.
- The slots represent equal logarithmic ranges in residual factors. This is proper because the MEDIAN model performs an additive fit to the logarithms of the system and problem factors.
- Each set of data is scaled individually, so readers should consider the numeric values of residual factors in comparing different compilation systems.
- Very small factors indicating test problems executing *much faster than expected* may indicate measurement noise — a test problem which a system actually optimized into a null but which was measured as having a small positive value would easily be orders of magnitude faster than the model would predict. Such test problems can distort the histogram by forcing it to represent a much larger range of factors than should be necessary.

The actual factors in each histogram are the residual values computed by MEDIAN.

7 JAN 1988 16:19:18

The histogram of ln residual values should be roughly Gaussian (normal distribution --- the familiar bell-shaped curve) with a mean of zero if the modeling assumption is correct. There are five fields in the output, as follows:

1. Slot number - range is scaled to always print 51 slots.
2. Actual factor associated with the beginning of slot
3. A one character indicator field flags residual values, using the following codes:
 - '<' Indicates a value less than low_outer fence.
Corresponds to more than two standard deviations below the mean. Much faster than expected.
 - '-' Indicates a residual value between low_inner and low_outer fences. Corresponds to between one and two standard deviations below the mean. Faster than expected.
 - ' ' Indicates residual value between low_inner and high_inner fences. Corresponds to values within one standard deviation of the mean. About as expected.
 - +' Indicates a residual value between high_inner and high_outer fences. Corresponds to between one and two standard deviations above the mean. Slower than expected.
 - '>' Indicates residual value greater than high_outer fence.
Corresponds to two standard deviations above the mean.
Much slower than expected.
4. A count of the number of entries in the slot.
5. A string of '*'s proportional to the number of entries in the slot, scaled so that the slot with the highest number of hits will have histogram_width stars.

7 JAN 1988 16:19:18

THE ENTIRE DATA SET

SLOT NUMBER	ACTUAL FACTOR	ENTRIES		HISTOGRAM
		IND	IN SLOT	
0	0.32	<	0	
1	0.34	<	0	
2	0.35	<	0	
3	0.37	<	0	
4	0.39	<	0	
5	0.40	<	0	
6	0.42	<	0	
7	0.44	<	0	
8	0.46	<	0	
9	0.48	<	0	
10	0.50	-	0	
11	0.53	-	0	
12	0.55	-	1	*****
13	0.57	-	0	
14	0.60	-	0	
15	0.63	-	1	*****
16	0.66	-	0	
17	0.69		0	
18	0.72		0	
19	0.75		1	*****
20	0.78		0	
21	0.82		0	
22	0.86		0	
23	0.90		2	*****
24	0.94		0	
25	0.98		3	*****
26	1.02		3	*****
27	1.07		0	
28	1.12		2	*****
29	1.17		0	
30	1.22		0	
31	1.28		0	
32	1.33		0	
33	1.39		1	*****
34	1.46		0	
35	1.52	+	0	
36	1.59	+	1	*****
37	1.66	+	0	
38	1.74	+	0	
39	1.82	+	0	
40	1.90	+	0	
41	1.99	+	0	
42	2.08	>	0	
43	2.17	>	0	
44	2.27	>	0	
45	2.37	>	0	
46	2.48	>	0	
47	2.59	>	0	
48	2.71	>	0	
49	2.83	>	0	
50	2.96	>	0	
51	3.10	>	1	*****

7 JAN 1988 16:19:18

INTEL8086_P

SLOT NUMBER	ACTUAL FACTOR	ENTRIES		HISTOGRAM
		IND	IN SLOT	
0	0.32	<	0	
1	0.34	<	0	
2	0.35	<	0	
3	0.37	<	0	
4	0.39	<	0	
5	0.40	<	0	
6	0.42	<	0	
7	0.44	<	0	
8	0.46	<	0	
9	0.48	<	0	
10	0.50	-	0	
11	0.53	-	0	
12	0.55	-	1	*****
13	0.57	-	0	
14	0.60	-	0	
15	0.63	-	0	
16	0.66	-	0	
17	0.69		0	
18	0.72		0	
19	0.75		0	
20	0.78		0	
21	0.82		0	
22	0.86		0	
23	0.90		0	
24	0.94		0	
25	0.98		1	*****
26	1.02		1	*****
27	1.07		0	
28	1.12		0	
29	1.17		0	
30	1.22		0	
31	1.28		0	
32	1.33		0	
33	1.39		1	*****
34	1.46		0	
35	1.52	+	0	
36	1.59	+	0	
37	1.66	+	0	
38	1.74	+	0	
39	1.82	+	0	
40	1.90	+	0	
41	1.99	+	0	
42	2.08	>	0	
43	2.17	>	0	
44	2.27	>	0	
45	2.37	>	0	
46	2.48	>	0	
47	2.59	>	0	
48	2.71	>	0	
49	2.83	>	0	
50	2.96	>	0	
51	3.10	>	0	

7 JAN 1988 16:19:18

I432_ADA_3

SLOT NUMBER	ACTUAL FACTOR	ENTRIES		HISTOGRAM
		IND	IN SLOT	
0	0.32	<	0	
1	0.34	<	0	
2	0.35	<	0	
3	0.37	<	0	
4	0.39	<	0	
5	0.40	<	0	
6	0.42	<	0	
7	0.44	<	0	
8	0.46	<	0	
9	0.48	<	0	
10	0.50	~	0	
11	0.53	~	0	
12	0.55	~	0	
13	0.57	~	0	
14	0.60	~	0	
15	0.63	~	0	
16	0.66	~	0	
17	0.69		0	
18	0.72		0	
19	0.75		1	*****
20	0.78		0	
21	0.82		0	
22	0.86		0	
23	0.90		0	
24	0.94		0	
25	0.98		1	*****
26	1.02		1	*****
27	1.07		0	
28	1.12		0	
29	1.17		0	
30	1.22		0	
31	1.28		0	
32	1.33		0	
33	1.39		0	
34	1.46		0	
35	1.52	+	0	
36	1.59	+	0	
37	1.66	+	0	
38	1.74	+	0	
39	1.82	+	0	
40	1.90	+	0	
41	1.99	+	0	
42	2.08	>	0	
43	2.17	>	0	
44	2.27	>	0	
45	2.37	>	0	
46	2.48	>	0	
47	2.59	>	0	
48	2.71	>	0	
49	2.83	>	0	
50	2.96	>	0	
51	3.10	>	1	*****

7 JAN 1988 16:19:18

NS16000_P_10

SLOT NUMBER	ACTUAL FACTOR	ENTRIES IN		HISTOGRAM
		IND	SLOT	
0	0.32	<	0	
1	0.34	<	0	
2	0.35	<	0	
3	0.37	<	0	
4	0.39	<	0	
5	0.40	<	0	
6	0.42	<	0	
7	0.44	<	0	
8	0.46	<	0	
9	0.48	<	0	
10	0.50	-	0	
11	0.53	-	0	
12	0.55	-	0	
13	0.57	-	0	
14	0.60	-	0	
15	0.63	-	1	*****
16	0.66	-	0	
17	0.69		0	
18	0.72		0	
19	0.75		0	
20	0.78		0	
21	0.82		0	
22	0.86		0	
23	0.90		0	
24	0.94		0	
25	0.98		1	*****
26	1.02		1	*****
27	1.07		0	
28	1.12		1	*****
29	1.17		0	
30	1.22		0	
31	1.28		0	
32	1.33		0	
33	1.39		0	
34	1.46		0	
35	1.52	+	0	
36	1.59	+	0	
37	1.66	+	0	
38	1.74	+	0	
39	1.82	+	0	
40	1.90	+	0	
41	1.99	+	0	
42	2.08	>	0	
43	2.17	>	0	
44	2.27	>	0	
45	2.37	>	0	
46	2.48	>	0	
47	2.59	>	0	
48	2.71	>	0	
49	2.83	>	0	
50	2.96	>	0	
51	3.10	>	0	

4.2.3 Residual Data

"Residual data" refers to the matrix of residuals, which flags outliers with a one column indicator symbol. The last line of the matrix, labeled "System Factors," gives a user the relative performance of all of the systems. Performances are presented as ratios to the first system, which is arbitrarily assigned a factor of one. Faster systems will be reflected as smaller numbers; slower systems will have larger numbers. In the sample data, system I432_ADA_3 is almost 14 times slower than system VAX_P_VMS. The indicator symbols, equations used, mnemonic symbols used, and any other useful explanation of the data which follows are described on the descriptor page which precedes the residual data. For examples, see the sample data which follows.

In the sample data, the residuals show that the problem ACKER (a version of Ackermann's function) is treated very differently by the different systems. This probably represents the different approaches taken to subprogram linkage.

The display of the residual table includes an additional row and column. The additional row at the bottom of the table contains the SYSTEM_FACTORS. The additional column at the right hand side of the table contains the PROBLEM_FACTOR for each test problem. Refer to Section 6.2 for a definition of problem and system factors.

7 JAN 1988 16:19:18

This table lists the problem, systems, and residual factors.
Values of factors are calculated to best fit the equation:

```
Time (systems, problem) ~=  
  Systems_factor(systems) * Problem_factor(problem);
```

and the residual factor is defined to be:

```
Time (systems, problem) /  
  Systems_factor(systems) * Problem_factor(problem);
```

The residual factor will be large when a system is slower than expected, based on the average speed of that system over all problems, and the average speed of that problem over all systems. Values less than 1.0 correspond to problems where the system is faster than expected.

A one character indicator field flags residual values, using the following codes:

- '<' Indicates a value less than low_outer fence.
Corresponds to more than two standard deviations below the mean. Much faster than expected.
- '-' Indicates a residual value between low_inner and low_outer fences. Corresponds to between one and two standard deviations below the mean. Faster than expected.
- ' ' Indicates residual value between low_inner and high_inner

fences. Corresponds to values within one standard deviation of the mean. About as expected.

'+' Indicates a residual value between high_inner and high_outer fences. Corresponds to between one and two standard deviations above the mean. Slower than expected.

'>' Indicates residual value greater than high_outer fence. Corresponds to two standard deviations above the mean. Much slower than expected.

Mnemonics are used in the output to qualify the REASON a certain test problem failed. The mnemonics are defined as follows:

CMP_TIME	Failed during compilation
DEPENDENT	Failed testing a system dependent feature
NO_DATA	Test not run.
NULL_TIME	Zero time; same as a no-op or a null stmt
PACKAGING	Test not available due to packaging. It may work, but may be included in another test which failed.
RUN_TIME	Failed during execution
UNRELIABLE	Test ran; data considered unreliable.
WITHDRAWN	Test problem from earlier release was withdrawn
DELAY_PROBLEM	Test problem included DELAY statements making it inappropriate to compare execution times in MEDIAN

7 JAN 1988 16:19:18

TEST PROBLEM NAME	VAX_P_VMS	INTEL8086_P	I432_ADA_3	NS16000_P_10	PROBLEM FACTOR
SEARCH	0.90	1.41	0.75	1.11	1.55
SIEVE	1.11	0.98	0.99	1.01	233.47
PUZZLE	0.89	1.02	1.01	0.99	12.99
ACKER	1.62+	0.55-	3.10>	0.62-	6.09

	VAX_P_VMS	INTEL8086_P	I432_ADA_3	NS16000_P_10
SYSTEM FACTORS	1.00	3.33	13.78	0.93

4.2.4 Statistical Summaries

MEDIAN computes several statistical summaries to observe the spread of the data and the goodness of fit of the model to the data.

The fences (which are the values used to determine the outliers to flag) and the standard deviations are printed. A Kolmogrov-Smirnov (KS) test is applied to the residual data,

comparing it to a normal (Gaussian) distribution. The output lists the k_plus and k_minus statistics of the KS test — for a discussion of the KS test and the significance of these statistics, refer to a statistics text, such as Introduction To the Theory of Statistics, by A. Mood and F. Graybill. A coefficient of determination test is also performed on the residual data to observe how much of the variation is explained. This statistic is analogous to the coefficient of determination test for regression models.

These summaries are of more limited interest than the others. The KS and coefficient of determination tests refer to how well the statistical model fits the observed data. The results of these tests are of most interest to persons considering alternate statistical models. Most users will not be greatly concerned. If they indicate a particularly bad fit, there may be anomalies in the data. The fit will not be good when comparing targets with vastly dissimilar relative performance characteristics, such as when one has software simulated floating point and the other has hardware floating point. Great care needs to be taken in comparing overall system factors between such systems, because the relative performance of the two systems will strongly depend on the workload (a processor with software floating point may compare favorably when no floating point operations are being performed). Readers may wish to re-analyze data using subsets which exclude problems not similar to their expected usage.

For the sample data, the KS test does not indicate a high probability of a match, which is not atypical (the KS test is a sensitive test). The coefficient of determination test states that the model explains 97% of the variation, which is good and typical for timing data.

7 JAN 1988 16:19:18

Statistical summary data on goodness of fit of the product model to the observed data. Outliers are flagged based on heuristics from Applications, Basics, and Computing of Exploratory Data Analysis by Velleman and Hoaglin, Wadsworth, Inc., 1981. For purposes of comparison, the cutoff values which would be obtained by using 'mean +/- sigma', and 'mean +/- 2 * sigma' are printed. Numeric values will differ somewhat. Readers may want to consider both sets when examining the residual data.

Data includes statistics comparing data to a Normal Distribution using a Kolmogorov-Smirnov test. For a good fit, the computed values of the k_plus and k_minus statistics should be small. If these values are large, refer to the Reader's Guide.

INNER FENCES	0.66~	1.51+
OUTER FENCES	0.48<	2.07>
exp(mean +/- sigma)	0.70	1.54+
exp(mean +/- 2 * sigma)	0.47<	2.28>

Test residuals for Normal Distribution

Kolmogorov-Smirnov statistics

number of points = 16

k_plus, k_minus 0.9758 0.6378

The probability that the residuals are from
the normal distribution is approximately 0.1489

This statistic is an overall measure of goodness
of fit based on the percentage of explained variance,
modeled after regression analysis of variance.

The maximum is 100%; the minimum is 0%. If the value
is small, refer to the Reader's Guide.

Source of Var Sum of Squares

explained 75.20
residual 2.35

total 77.55 n = 16

Percentage of variance explained = 96.97%

4.2.5 Residual Data Summary

The residual data summary is a summary of the extremes, quartiles, and median, in factor and log form for the entire set of residuals. As in the residual matrix, outliers are flagged with a symbolic indicator (which is described on the descriptor page which precedes the data). The data in this matrix is for all systems combined, hence the label "COMBINED" in the printout. Each of the columns of the printout are described prior to the display of the data. The spread is defined to be upper quartile / lower quartile. See the sample data which follows.

7 JAN 1988 16:19:18

Display 'box' summary statistics for

A name identifying what data is being displayed

1. The whole set of residual data

2. Problems where a quartile was far from mean - and name problem

3. Each system

The summary lists:

1. The minimum value

2. The lower quartile - 25th percentile

3. The median - 50th percentile

by construction - iterative Polish cycling - should be close

to zero

4. The upper quartile - 75th percentile

5. The maximum value.

6. For actual factors, print (upper quartile / lower quartile).

Where the spread is large, there was a large variation in the performance achieved. Problems which show a large spread discriminate between systems.

SUMMARY DATA FOR ALL RESIDUALS

	MIN	LOWER	MEDIAN	UPPER	MAX	SPREAD
COMBINED	0.55-	0.90	1.00	1.11	3.10>	1.23

SUMMARY DATA FOR THE LN OF ALL RESIDUALS

	MIN	LOWER	MEDIAN	UPPER	MAX
COMBINED	-0.60-	-0.10	0.00	0.10	1.13>

4.2.6 Problem Name Outliers

Outliers are values which are far from the normal, or expected distribution. In the case of the following data, outliers refer to those values outside the inner (and outer) fences. The descriptor page which precedes the residual data summary also applies to the problem name outliers. See the following sample data.

This summary prints the extremes, quartiles, and median, in factor and log form for test problems where at least one of the quartiles is flagged as an outlier.

For the sample data, the only problem so flagged is ACKER, which shows a large variation in relative performance between systems.

7 JAN 1988 16:19:18

SUMMARY DATA FOR EXCEPTIONAL PROBLEMS

TEST PROBLEM NAME	MIN	LOWER	MEDIAN	UPPER	MAX	SPREAD
----------------------	-----	-------	--------	-------	-----	--------

ACKER | 0.55- 0.62- 1.00 1.62+ 3.10> | 2.61

LN SUMMARY DATA FOR EXCEPTIONAL PROBLEMS

TEST PROBLEM NAME	MIN	LOWER	MEDIAN	UPPER	MAX
ACKER	-0.60-	-0.48-	0.00	0.48+	1.13>

4.2.7 System Summary

The system summary contains the extremes, quartiles, and median, in factor and log form for each system. The number of test problems in various categories on which the system factors were computed is also displayed.

A system which executed 5 tests out of 2,000 and did those 5 test problems well will be more favorably reflected in the MEDIAN output than one which executed 1,990 tests out of 2,000 moderately. For this reason, the last summary is provided which summarizes the tests each system ran and indicates the percentage of tests of the ACEC that were run. The categories of the measurements are provided to indicate which are used in the statistics performed by MEDIAN.

The column labels of the last summary utilize the mnemonics used by MEDIAN when printing the raw data matrix and the residual data matrix. In this last summarization, the column labeled "VALID TIMES" corresponds to the measurements of those test problems which executed successfully. The "NULL TIMES" column corresponds to the "NULL_TIME" mnemonic; the "UNRELIABLE TIMES" column corresponds to the "UNRELIABLE" mnemonic; and the column labeled "OTHER" accounts for the rest of the mnemonics. Only the measurements which fall under the "VALID TIMES" column are used in the MEDIAN analysis. Total number of tests attempted is printed, and finally a percentage is printed. This percentage is calculated as:

$((\text{valid times} + \text{null times} + \text{unreliable times}) / \text{total num of tests}) * 100.0$

7 JAN 1988 16:19:18

SUMMARY DATA FOR EACH SYSTEM

SYSTEM UNDER TEST	MIN	LOWER	MEDIAN	UPPER	MAX	SPREAD
VAX_P_VMS	0.89	0.90	1.00	1.11	1.62+	1.23
INTEL8086_P	0.55-	0.98	1.00	1.02	1.41	1.03
I432_ADA_3	0.75	0.99	1.00	1.01	3.10>	1.01
NS16000_P_10	0.62-	0.99	1.00	1.01	1.11	1.01

LN SUMMARY DATA FOR EACH SYSTEM

SYSTEM UNDER TEST	MIN	LOWER	MEDIAN	UPPER	MAX
VAX_P_VMS	-0.12	-0.10	0.00	0.10	0.48+
INTEL8086_P	-0.60-	-0.02	0.00	0.02	0.34
I432_ADA_3	-0.28	-0.01	0.00	0.01	1.13>
NS16000_P_10	-0.48-	-0.01	0.00	0.01	0.10

SYSTEM UNDER TEST	VALID TIMES	NULL TIMES	UNRELIABLE TIMES	OTHER	TOTAL NUM OF TESTS	PERCENT
VAX_P_VMS	4	0	0	0	4	100.00%
INTEL8086_P	4	0	0	0	4	100.00%
I432_ADA_3	4	0	0	0	4	100.00%
NS16000_P_10	4	0	0	0	4	100.00%

4.3 SINGLE SYSTEM ANALYSIS

The ACEC single system analysis (SSA) tool is designed to help extract the information implicit in the relationships between related test problems. It analyzes the measurements obtained from executing the ACEC test suite on one system by comparing related test problems. Some relationships between problems include:

- Some are optimizable and hand-optimized versions of a problem.
- Some perform the same operations using different coding styles.
- Some are versions with and without specification of certain pragmas, such as SUP-PRESS or INLINE.

These comparisons add no information not present in the "raw" performance data. However, with over one thousand separate test problems, comparing the related test problems by hand is a time-consuming task, particularly for anyone not initially familiar with the relationships between the test problems. The SSA tool knows the relationships between test problems, and the report it generates highlights the significant results.

The MEDIAN tool displays how one system performs relative to other systems. The SSA tool compares results from related test problems executed on the same system. Roughly speaking, the MEDIAN tool provides data most useful for selecting between different compilation systems while the SSA tool provides data to help programmers efficiently use a compilation system after it has been selected.

The SSA report can be used for comparisons between systems by manually examining two (or more) reports on a table by table basis. This would be tedious if complete reports were being perused, but could be very useful for someone with a tightly focused interest.

To elaborate on the differences between the SSA tool and the MEDIAN tool, consider two test problems, one of which is a hand-optimized version of the other. When either all sample systems perform the optimization or none of them do, the MEDIAN residuals for both problems might not flag any system as anomalous. An examination of the MEDIAN residual matrix will not tell the ACEC reader whether or not any system optimized the optimizable problem. This information is precisely what the SSA tool will provide. Both types of information can be valuable — if there are no performance differences between the different systems, then for a source selection activity, the systems are comparable and need not be distinguished. Programmers writing code for a particular system may want to know whether a particular optimization is performed — unless they are concerned with portability, they may not care whether the optimization is performed by any other compilation system.

By reporting whether specific optimizations are performed, the SSA tool permits a programmer to change coding styles to adapt to the strengths and weaknesses of a system. A programmer who knows that a system performs no loop invariant motion will know that using temporary variables and performing loop invariant motion "by hand" in the source text can be a profitable operation. With an optimizing compiler, such a coding style may be superfluous and duplicate effort that the compiler will perform by itself. There may be some systems where record processing is faster when coded as a sequence of operations on each component of the record. On such a system, a programmer would minimize the use of record operations in time-critical code. Other areas where the SSA tool may highlight potential performance problems include: aggregate processing; package elaboration; subprogram linkages; passing of unconstrained parameters; tasking constructions; exception processing; block entry; etc.

Knowing the extent of optimization from the SSA can help in evaluating some vendor claims for enhancement possible in future versions of a compiler. If the evaluated compiler does no detectable optimizations, a vendor claim that a future release will generate much better code is believable – they have ample room for improvement.

The outline of the SSA report is as follows:

- Table of Contents

This section lists the individual findings and the page number they may be found on. If there is not sufficient data to permit a finding to be made, there will be no entry in the table of contents for it. The order of presentation will be the same for every analysis.

- Findings

The major report categories are

- * Language Feature Overhead
- * Optimizations
- * Runtime System Behavior
- * Coding Style Variations
- * Ancillary Data
- * Failure Analysis Report
- * Code Size Report
- * Compile Speed Report

- Summary report

This will briefly summarize the findings in the first four major report categories.

- Missing Data Report

This is an optional section which lists the test problems for which data was not available.

The actual report consists of four files, corresponding to each section listed above. Each of these will be discussed and illustrated in turn.

4.3.1 Table of Contents

The table of contents lists the groups and the page number in the main report where the relationships between the problems in the group are presented.

An example table of contents would look like:

Optimizations	
Algebraic Simplification - Integer	page 12
Coding Style	
Array Assignment	page 23

Figure 2: Example of a Partial Table of Contents Page

4.3.2 Main Report

In the first four categories listed above, Language Feature Overhead, Optimizations, Runtime System Behavior, and Coding Style Variations, there will be a table printed for each group of related test problems. This will present the names and a brief description of the problems. There are two types of presentations, discussed separately.

Users may analyze subsets of test problems and so not print tables for problems not of interest (or where results are not available). This is done by leaving data out of the input data files read by the SSA program.

4.3.2.1 Multiple Comparisons

These are used when there are several related test problems, sometimes reflecting the same example coded in different styles.

Each problem is named and briefly described, and the execution times (and, if available, code sizes) are presented for them.

The report will display sets of test problems whose results do not have a statistically significant difference. This is based on the magnitude of the estimates for each problem and the observed variation in each problem. When there is no statistically significant difference between test problem results, *there is no performance reason for preferring one alternative to another*. When there are statistically significant differences, the magnitude of the differences may not be large enough to justify modifying coding style for performance reasons.

In cases where the ACEC user has had to copy results — say from an embedded target without an upload capability where these results would have had to be typed in by hand — complete data may not be available. In such a case, the SSA may not have enough information to perform the similar group comparisons; however, it will perform the rest of the analysis and print as much of the report as it has information to do.

Coding Style			
Array Assignment			
Test Name	Execution Time	Bar Chart	Similar Groups
ss77	17.4	*****	
ss78	17.5	*****	
ss388	39.5	*****	
ss80	57.1	*****	
ss79	missing		
Code Size			
ss77	86	*****	
ss78	90	*****	
ss80	102	*****	
ss388	135	*****	
ss79	missing		
Individual Test Descriptions			
ss77	e1 := (1 .. 10 => 1.0) ;		
ss78	e1 := (1.0, . . . 1.0) ;		
ss79	e1 := xe1 ;		
ss80	FOR i IN 1..10 LOOP e1 (i) := 1.0; END LOOP;		
ss388	e1 (1) := 1.0 ; . . . e1 (10) := 1.0 ;		

Figure 3: Multiple Comparison Table Example

This layout is straightforward. The header presents the type and name of the group (in this example, it is a set of problems reflecting different coding styles for array assignment). The next section contains column headers for the individual test problem names; their execution time in microseconds (problems for which timing data is unavailable will be flagged as such); a graphic representation of the execution time is presented to make simple visual comparison easy, permitting a reader to judge whether two problems are roughly comparable without having to examine the numeric values — they could “flip” through the main section of the report looking for groups where large differences show up; and a similar groups column.

4.3.2.2 Statistically Significant Differences

Any measurements which are subject to error (as are the timing results from the ACEC) must be interpreted with care. If we know that time A is 10.0 and that time B is 20.0, we do not know whether we can safely say that time B is twice as large as time A, or even if it is safe to say that time B is larger than time A, unless we know how large the measurement error is. If the error magnitude is less than 0.1, then both of those conclusions are safe. If the error magnitude is greater than 100.0, then both of those conclusions are obviously false (not only can we not say that B is larger than A, we cannot even say that A and B are nonzero times).

Statisticians have studied problems like this and we make use of their findings. Statistically significant differences are differences which can be assumed to be reliable and believable differences.

The statistical analysis uses Bonferroni's method for for multiple comparisons (refer to Chapter 3 of *Analysis of Messy Data: Volume 1*, by G. Milliken and D. Johnson, Van Nostrand Reinhold, 1986 for details). This method was selected because it permits all pair-wise comparisons and does not require equal sample sizes. The determination of similar groups requires knowledge of the number of samples and variation — in the case of the ACEC execution time measurements, the number of samples is the outer timing loop count and the variation is the observed standard deviation. In cases where the ACEC user has had to copy results — say from an embedded target without an upload capability where these results would have had to be typed in by hand — complete data may not be available. In such a case, the SSA may not have enough information to perform the similar group comparisons; however, it will perform the rest of the analysis and print as much of the report as it has information to do.

We do need to distinguish between statistical significance and practical significance. If the measurements are very accurate, then very small differences will be statistically

significant, that is, they will be real differences. However, the reader of these results may not care if one coding style is really one per cent faster. The rule for interpretation is simple: only look at differences which are statistically significant. The judgement as to whether these differences are large enough to be of practical importance will depend on non-statistical considerations, that is, it depends on your judgement.

The code sizes follow the same standard format except that there is no statistical analysis and therefore, no Similar Groups section. If code size data is available, it should be precise and there should be no measurement error. The unit of measurement is bytes (8 bits).

After this section, each test problem is briefly described.

4.3.2.3 Binary Comparisons

These are primarily used for detecting whether a particular optimization is performed or not, based on whether or not there is a significant difference between two problems where one is a hand-optimized version of the other.

Optimizations	
Algebraic Simplification : Integer	
Description	Optimized?
Time : ss50 (2.0) vs ss7 (2.0)	Yes
Size : ss50 (92.0) vs ss7 (92.0)	
ss50 => ii := ll ** 0 ;	
ss7 => kk := 1 ;	

Figure 4: Example of Main Report : Paired Comparisons Page

The column "Description" states the optimization whose presence the pair of problems is designed to detect. The "Optimized" column will represent whether the SSA believes that the optimization has been performed. Possible answers are YES, MAYBE, NO, and MISSING. The detection is based on the difference between the execution times of the two problems — the difference will be small where the optimization is performed, and large when it is not; but the interpretation of intermediate values can be unclear and will be represented by a MAYBE code.

4.3.2.4 Ancillary Data

Some of the performance tests produce ancillary information as a side effect of executing them. This data is collected by the SSA tool and presented in the SSA report. The types of information included in the ancillary data report are:

- The rates for the Whetstone and Dhrystone programs in statements per second.
- The time per rendezvous for the tasking tests which perform rendezvous.
- The time for one procedure call on ACKER1, ACKER2, and TAK.
- The observed numeric errors in the GAMM calculation for GAMM1 and GAMM2.
- Whether the RECLAIM test problems immediately reuse space.
- Some details about the treatment of arithmetic expressions.
- Size of packed data structures.
- Activation record size.
- Whether asynchronous I/O is performed.
- Details of task scheduling, including whether the system is using a run-till-blocked scheduler or time-slicing among equal priority tasks, and the quantum-size where appropriate, and whether the system waits until outstanding delay statements are completed before completing an aborted task.
- Whether generic instantiations are shared.
- Whether constraint checking is performed when pragma suppress is requested.
- And other miscellaneous information.

4.3.2.5 Failure Analysis Report

This section tabulates failures in two ways: by language features prone to failure and by reason for failure. The first table lists failures in generics, in Chapter 13 features, in

tasking, and in other categories. The second table lists failures because of compilation problems, execution time problems, and so on. This table may not be very helpful unless the user has entered the appropriate error codes for each failure. While some of these are generated automatically by the benchmark suite, many are not. For example, an execution time failure which causes a system crash will usually require that the user note the results and enter the proper code.

There is one inconsistency between this table and the Missing Data Report section of the SSA. The SSA *program* makes use of "unreliable" data since the statistical procedures supported take into account the larger variation in these measurements. Therefore, unreliable data does not appear in the Missing Data Report.

4.3.2.6 Code Size Report

This section summarizes the code size data, giving average bytes per line, as well as highest and lowest test problems. Lines here are physical lines which are included in the ACEC timing loop: between PRAGMA include("starttime") and PRAGMA include("stoptime0"). Parallel results are provided using bytes per semicolons. Finally data is presented for a selected group of larger test problems.

4.3.2.7 Compile Speed Report

The compilation speed analysis also uses both physical lines and semicolon counts (both based on complete files). Again, averages, extremes, and selected examples are printed. As mentioned in the User's Guide, Section "PREPARING THE DATA", it is required that the user gather and format the data, since there is no portable method for doing this chore. However, any user who wishes to analyze compile speed using the Median tool will already have done the work required for SSA analysis, provided the formatting guidelines given in the User's Guide, Section "SSA" are followed.

4.3.3 Report Summary

The summary section will present a condensation of the results of the first four major report categories in the main section.

Optimizations						
Yes	Maybe	No	Missing	No Stat	Total	Page
Algebraic Simplification: Integer						
3	1	1	1	1	7	10

Coding Style Variations				
Performance Range	Significant Difference?	Missing Tests	Total Tests	Page
Array Assignment				
17.4 .. 57.1	Yes	1	5	12

Figure 5: Example of a Report Summary

For the optimization comparisons, the summary will list the totals for YES / NO / MAYBE / MISSING / NO STAT / TOTAL for each optimization technique. For the multiple comparison, each group will be summarized by presenting the range of observed values, whether more than one statistically different group was detected within the problems, and the number of missing and total problems in the set.

4.3.4 Missing Data Report

The missing data report lists the test problems for which no timing measurements were reported and, if the appropriate error codes have been entered, the correct reason for the missing data.

Optimizations	
Algebraic Simplification : Integer	
ss51 is missing	err_at_compilation_time
Coding Style	
Array Assignment	
ss79 is missing	err_at_execution_time

Figure 6: Example of a Partial Missing Data Report

5 DETAILS OF TIMING MEASUREMENTS

This section describes how time and code expansion measurements are taken, the constraints placed on test problems to permit them to be measured, the sources of measurement errors, and the steps taken to minimize errors and the error bounds.

An ACEC user not familiar with computer performance tests may wonder why there needs to be any discussion about timing measurement, anticipating measurement code similar to:

```
start_timing := calendar.clock;  
test_problem_to_be_executed  
stop_timing := calendar.clock;  
elapsed_time := stop_time - start_time ;  
output_problem_name_and_elapsed_time
```

Before discussing the ACEC approach to measuring execution time, it is worthwhile to explore some of the reasons why this simple approach was not adapted.

- Precision:

The precision of the type CALENDAR.TIME returned by the CALENDAR.CLOCK function is implementation dependent. It is typically in the range from one to fifty milliseconds, but on one system was one second. The precision of the clock measurement will introduce quantization errors. Where one clock tick is a large fraction of the measured difference between START_TIME and STOP_TIME, the calculated estimate for problem execution time cannot be considered very precise.

The standard benchmarking technique to deal with this concern is to execute a test problem multiple times so that the execution time will be large enough that the quantization errors are acceptably small. Then the measured time is divided by the number of iterations. The overhead of a null loop is then subtracted off and an estimate for the execution time of a single iteration is reported.

Where no consideration has been given to this question, the results will contain an unknown implementation dependent quantization error. Measurements of "fast" test problems will contain many zero values where the clock did not change between starting and stopping measurement.

- Economy:

In an attempt to compensate for the precision problem discussed above, a developer might explicitly code in looping factors to the test problems to insure that

they execute for a "sufficiently long" time. However, there is a large amount of variability between different compilation systems. The source which executes in ten seconds (a reasonable time for obtaining measurement on many systems) on one compilation system may execute a thousand times faster or slower on another system. Due to different code generation techniques, optimizations and target hardware characteristics, the variations are not uniform between all test problems on a system.

In an effort to avoid test problems which don't run long enough to be measured reliably, it is easy to develop test problems which run for many hours. This is an uneconomical use of computer resources which is particularly distressing because using more sophisticated measurement techniques will permit comparable accuracy with much less elapsed time and much less use of computer resources.

- Error estimates:

The naive approach will give no estimate of the repeatability of the measurements. As a physical process, clock readings should be expected to be subject to random variations. Statistical arguments would suggest that the variability in measurements should be observed and reported. At a minimum this would suggest calculation of sample standard deviation. Calculation of confidence levels for measurements are possible and would be reassuring. Neither statistical concept is supportable with the simple measurement methodology.

- Optimization:

Ada permits compilers to generate code which executes some statements in a different order than presented in the source text. Using the naive approach, it is possible that an optimizing compiler might move test problem code outside the calls on the clock function. This is a potential problem for any measurement technique, but using design approaches which explicitly consider the problem and try to allow for it is more likely to be successful than the straightforward approach which ignores the issue.

5.1 REQUIREMENTS OF THE TIMING LOOP CODE

The timing loop code is designed to satisfy several requirements. This section of the guide contains detailed information about error analysis and considerations concerning construction of timing measurements code. Readers without a statistical background may wish to skim this section.

1. It shall produce accurate measurements. It attempts to satisfy predetermined error tolerances and confidence levels.

2. Its output shall show the variations in the observations. That is, the standard deviation of the measurements.
3. Sufficient information shall be displayed to permit interested users to compute other standard statistical functions. A user will have sufficient information to compute a z-statistic (refer to statistics textbook, such as Introduction To the Theory of Statistics by A. Mood and F. Graybill, McGraw-Hill, 1963, where it is discussed in the chapter on the Central-Limit Theorem, page 151).
4. The timing loop code shall be portable. Its execution shall not require special test equipment, system dependent calls, or operator interaction. It will not need to be manually tuned to the properties of the target system. In particular, it shall not be necessary to modify a system parameter based on the speed of the target machine, either making it larger to produce a measurement with sufficient accuracy, or making it smaller to cut down on elapsed time to perform the tests.
5. The timing loop code shall not require the support of integer types with more than 16 bits of precision.
6. The timing loop code shall not assume a system clock which is precise to the level of a few machine instructions.
7. The timing loop code shall test for jitter (small random variations in the clock readings) and try to compensate for it. The compensation is achieved by requiring each measurement to execute for a minimum elapsed time — long enough so that the number of clock ticks will "average out" the random jitter.
8. The timing loop code can be adapted to compilation systems which defer code generation until link time. This can be difficult, because a system which defers code generation will have access to information about the entire Ada program, and can apply optimizations across compilation units. This can circumvent the standard techniques to thwart optimizing compilers, making it hard to insure that a test problem is actually executed once for every iteration of the timing loop. Such an optimizer can perform detailed analysis to determine that a test problem is actually a no-op.

Calling assembler coded subprograms should be sufficient to thwart deferred code generators. Assembler routines can perform arbitrary operations, including reading and writing blocks of data between memory and I/O devices. A compiler must assume that a called assembler routine could reference and/or modify any externally visible variable. For a compiler to analyze an arbitrary assembler routine and determine correctness-preserving optimizing transformations is infeasible. An assembler subprogram could branch to an absolute address, presumably in the operating system, whose operations are not available to the Ada compilation system.

An arbitrary assembler routine might be self modifying, and a compiler must assume that it is. Including a call on an assembler routine in the timing loop should be sufficient to thwart unintended optimizations by compilation systems using deferred code generation. A compilation system which does not support interface to assembler coded subprograms, and does deferred optimization, will make it difficult to collect measurements. It is expected that serious Ada compilation systems for Mission Critical Computer Resource (MCCR) applications will provide the capability to link to assembler code, so that the applications can access unique I/O and processor dependent features only visible to assembler coded routines.

9. The timing loop code shall minimize the use of computing resources where this does not compromise the other requirements listed above.
10. When the time to execute a test problem is not converging to stable values, the timing loop shall not increase the number of repetitions (or cycle) excessively in the vain hope that it will eventually converge. In particular, the elapsed time speed on any individual test problem (which does not fall into an infinite loop due to errors in the implementation) should be limited.

After a test problem has executed for an excessive time and the timing loop regains control, even if the measurements have not converged to the confidence level requested, execution should stop. The variable GLOBAL.EXCESSIVE_TIME controls this cutoff and in the distributed ACEC has a value of 30 minutes. This value is adaptable by the user.

After a test problem has executed for a smaller but still large time, print a message to the results file. It is reassuring when running the test suite interactively to know that the program has not fallen into an infinite loop.

5.2 A PRIORI ERROR BOUND

The error tolerances computed by the software vernier determine an error bound on the test problems. The technique for testing these bounds is described in detail in Section 5.4.1. The significant point to remember in this section is that the ACEC does try to insure that the test problems are correct in the sense that the timing measurements satisfy predetermined error tolerances.

By specifying desired tolerances before executing the test problems, the ACEC fixes a priori goals for measurement accuracy.

The timing loop code measures time by referencing the system clock. Clock readings are subject to the usual statistical variations associated with physical measurements,

and can be expected to show random variations. Assuming that the errors are normally distributed, there are statistical techniques to estimate the accuracy of the estimates achievable in the presence of such random errors.

There are some systematic sources of potential errors which are not amenable to statistical methods for compensation. The methods used to detect the presence of such sources are discussed in their own section.

5.2.1 Random Errors

The timing loop displays the standard deviation of the observations, and an indicator if the predetermined confidence level was not achieved. Refer to Section 5.4.1. Timing measurements are subject to errors; the statistical fluctuations in measurements is an indicator of the variability in the underlying process.

The test problem times being measured are usually much smaller than the granularity of the system clock. The ACEC works around this problem by using a "timing loop" which repetitively executes the Ada problem to be measured. The time to execute the timing loop overhead is subtracted out of each measurement.

The ACEC uses a "dual loop" approach to measuring test problems. It calculates the time to execute a test problem by executing the test multiple times (to make the total execution time large enough to accurately measure with the relatively coarse clocks available for program use) and subtracts out the overhead introduced by the looping structure. As part of the initialization in each test program (INITTIME), the ACEC computes the execution time of the null control loop. This precomputed time is used to calculate each test problem time. The ACEC initialization code measures the null loop several distinct times in order to observe possible variations in the control loop.

The ACEC minimizes the effect of small random errors by requiring that the test problem execute for more than some minimal time before accepting a measurement. See the discussion of the clock vernier in Section 5.4.1 for details. This minimum time is chosen to be large enough that random errors (jitter) observed during program initialization by INITTIME will not contribute more than an additional one per cent error to the measurement. The number of iterations performed is printed, labeled "inner loop count."

The magnitude of remaining random errors is estimated by, after determining the appropriate number of iterations of the test problem, repeating the measurement for multiple cycles. The timing loop code uses standard statistical techniques (Student t-test) to determine the proper number of cycles to perform so that the sample mean will be within a predetermined tolerance level of the population mean with a predetermined confidence level. The per cent standard deviation of the measurements is printed, labeled "sigma."

A large variation indicates a measurement where substantial errors may be present. If the timing loop was not able to satisfy the stated confidence level, an error indication (a “#” character) is also output. The cycle count is printed as the “outer loop count.”

Measurement of test problems which have been optimized into null statements could produce small non-zero values for a minimum time because of noise in the timing loop. The ACEC checks for this possibility before printing a result. If the code expansion size measurement is zero it will print the time as zero. This is not a major problem, but because the MEDIAN summary lists the number of NULL test problems, it would be nice if the number of null problems reported is constant between runs of the same executable ACEC test program.

If the measured problem time is less than the precomputed null loop time, a meaningless negative execution time could be reported for the test problem. This condition is explicitly tested for in the ACEC timing loop and appropriate compensating action taken:

- When the difference between the null loop and the measured test problem is small (that is, less than the variations observed in the initialization of the null timing loop) then the difficulty is assumed to be due to noise in the measurement process and the calculated time is replaced with a zero measurement. Replacing small negative values with zero improves the cosmetic appearance of the output and is necessary because the analysis programs consider negative values to be special codes used to indicate the reasons for problem failures.
- When the difference between the null loop and the measured test problem is large, the difficulty is assumed to be more serious. An unreliable measurement result is printed. This condition can (most often) occur when there are contending tasks in the system, which were executing when the null loop was initialized and terminated when the actual test problem was run. It can also occur, as discussed in Section 5.2.2, when the instructions generated to execute the timing loop in the null loop are significantly longer than in a test problem.

A careful ACEC user should examine the test problems which execute in zero time.

5.2.2 Systematic Errors

If the code generated for the timing loop for the NULL statement takes a different amount of execution time than for a general test problem, the calculated performance estimate for the test problem will be erroneous. Such an error cannot be compensated for by performing additional iterations since it is not a random term which can be “averaged

out." It is a systematic measurement error. Listed below are potential causes for such variations, and the steps taken in the timing loop code to minimize their impact:

1. Systematic errors would result if different instructions were generated for the timing loop code for measuring the *NULL* statement than for *general test problems*. An optimizing compiler which keeps the timing loop control variables in registers for the *NULL* statement might not be able to keep them in registers in general. This would result in different instructions for the *NULL* loop and for the test problem, and would invalidate the timing hypothesis. If this were to happen, complex test problems will be reported as being slower than they really are, because the measured times would include the extra time (relative to the null timing loop) to maintain the loop control variables in memory. To prevent this from happening, the timing loop most often uses *library scope variables*. Values for library scoped variables must be assumed to be modified by external subprograms, making values for library scope variables saved in registers unreliable across subprogram calls where the test includes an external subprogram call. The timing loop contains a conditional call on an external subprogram to "break" register allocation. Making the external call conditional depending on external variables should inhibit compile time optimizers. By arranging for a false condition so that the subprogram is not called, the time to execute the null loop will be minimized. Not calling a subprogram will decrease the variability in execution time of the timing loop code between the null loop and "real" test problems on target machines with memory caches because the cache will not be "flushed" by loading the code for the external procedure into the cache on each iteration of the test problem. There could be a discrepancy introduced here if the test code always called a procedure and, depending on the alignment of the test problem and the procedure relative to the cache, calling on the procedure forced a cache fault in the null loop and not in the test problem (or visa versa). System sensitivity to cache alignments is a real effect (not induced by measurement artifact) which can make it risky to extrapolate timing measurements from small test problems to other usages. Minimizing execution time for the null loop can decrease the execution time of test problems in two ways: first (and least importantly) it will reduce the time to execute the null loop itself; second, it will increase the fraction of elapsed time spent executing the test problem reducing the overhead due to the null loop.

An example was encountered with the DEC Ada Version 1.0 system. The base address of a package was required in a register. For some problems, the base address was loaded outside the timing loop, and for others it was loaded on each iteration. The timing loop therefore had a variable number of instructions. The measured time to execute a (short) test problem including the timing loop code

was less than the measured time to execute the null timing loop as the code was generated in the null loop initialization code.

Of course, even if the same machine instructions are generated, they may take different times to execute due to machine architecture variations. These sources of variation are discussed later. It should be clear that when the instructions generated for the timing loop code are different, the time to execute the timing loop code can vary.

On target processors which have different formats (such as, long and short branch instructions) the timing loop code execution time for a large test problem will be longer than for the NULL loop. The computed time to execute a large test problem will be reported as longer than it should be because the branch in the timing loop code will now be a long format branch which will make the timing loop overhead longer than in the NULL loop time as computed during program initialization. For a large test problem, the difference in execution time due to a long format branch will usually be (relatively) small.

The approach of the ACEC to minimize this source of systematic errors has been to exercise care in the design and testing of the timing loop code. The initialization code in the timing loop tests the consistency of the null loop time, and large variations in the initialization code may suggest variations in the generated code. The most likely cause of variation is contending processes on the target system, but if contention can be eliminated as a source of variation, then investigation of alternate code may be appropriate. If the size of the NULL loop code varies between different test programs, compiled with the same suppression and optimization options, then the user has strong reason to suspect that different code is being produced. For some test problems on some target machines, anomalous results may still occur. It is hoped that this discussion will alert ACEC users to the potential problems in collecting measurements.

2. Establishment and maintenance of addressability to external packages can introduce unanticipated overhead. As has been seen in decades of experience with the IBM S/360 architecture, the efficiency of compilers in maintaining addressability is a major factor in determining overall performance. The amount of time it takes to execute a high order language statement such as "x:=y;" can vary greatly depending on whether base registers are available for "x" and/or "y." An optimizing compiler will preserve base registers across several statements. This effect is a reflection of a real and unavoidable aspect of target processors with base register addressing. What appears to a casual programmer to be irrelevant differences between statements occurring in different locations in a program can have major performance impacts. The ACEC includes a sample of large test problems to try to observe these effects.

There are many popular machine architectures where this particular complication in obtaining high performance code does not occur.

There are several test problems which reference data from multiple packages to observe possible addressability overheads. In particular, test problems SS469 through SS477 are designed to specifically study this issue.

The approach of the ACEC to minimize the effect this problem might have on the calculation of timing measurements has been to insure that the variables controlling the timing loop are in an external package "GLOBAL" and the timing loop contains a call on an external procedure, which potentially can modify all register settings.

3. Some compilers can generate different code for a construct depending on the nesting level where the construction occurred. The timing loop code surrounds each test problem with two levels of loop nesting. A compiler might keep the innermost **FOR** loop index in a register. Here, the time it takes to enter a **FOR** loop will depend on the nesting level of the **FOR** loops, because the nested loops must save and restore the registers for the outer loops. Predictable complications are avoided in the timing loop code by not using **FOR** loops for timing.

The approach of the ACEC to minimize this source of systematic errors has been to use library scope variables to control the execution of the timing loop.

4. Memory organization in processors can cause the timing loop to take variable amounts of time. The time to execute the timing loop can vary with the alignment of the code in the address space. Where the test problems being measured are of differing sizes, the beginning and end of the timing loop (STARTIME and STOPTIME0) will be in different relative locations. This can cause performance differences. Consider the following examples:

- On a virtual memory system, for two small test problems of the same size, the entire test problem code might be assigned to one memory page in one example and span two memory pages in the other. If a page fault occurs, the execution time will be orders of magnitude slower than when no page fault occurs. Assuming that the test problem is being executed with a sufficient working set size so that enough physical memory is available to keep the test problem loaded, one iteration through the test problem will be sufficient to load all the necessary virtual memory space into real memory. If there is not sufficient real memory, the ACEC test problems will measure a paging system while it thrashes, which is neither helpful nor interesting. An application with insufficient memory on a virtual memory system, either on the physical machine or in the allocation of real memory permitted by the target operating system, will run slowly. The way to fix the problem is well known and independent

of the use of Ada: give the program sufficient memory. An Ada compilation system may aggravate the problem by generating programs with large working sets! The ACEC explores this effect by examining the code expansion rates and the size of the RTS.

The ACEC timing loop will compensate for initial gross timing errors, caused by the initial page fault(s) to load the test problem into memory in three ways:

- * The inner timing loop will increase the test problem repetition count until the measured execution time for the test problem is greater than the value of "min jitter compensation." On many target systems, this time will be greater than the time to execute a small test problem *including* processing a page fault, so the normal timing loop control mechanism will be to increase the repetition count until the execution time for the test problem is sufficiently large.
- * STOPTIME0 will reset the inner timing loop whenever a new measurement is less than half or more than double the execution time of the current best estimate. When an initial execution is much longer than subsequent measurements due to processing page faults, the later measurements will observe a *much* faster apparent execution time. This will trigger an increase of the repetition count and a re-entry into the inner timing loop.
- * The outer timing loop reports the minimum time over several samples, which is the value used for later analysis. An initial measurement which is slower than prior readings (but not so much slower as to trigger the reset of the inner timing loop discussed above) will be discarded. If the variation in measurements when the first execution of the test problem includes page fault processing is small enough so that the measurements were within the requested confidence level anyway, the errors introduced by processing the memory faults were a small percentage of the total problem execution time.

The interested reader should refer to Section 5.3 for more discussion of the timing loop code.

Even when page faults do not occur, some virtual memory systems may incur some additional overheads in accessing memory pages, such as keeping track of the least recently used page.

- Cache memory organizations can introduce variability. Cache memory organizations typically partition physical memory into fixed size blocks and define a simple memory mapping from "main" memory to cache frames, using high order memory address bits. Several "main" memory blocks will map to the same cache frame, and the cache management hardware tracks the current contents of the frame and will load the "main" memory block into the cache

frame as memory references require. If a program was unlucky, it might find that the head and tail of the timing loop are allocated to "main" memory locations which map into the same cache frame: forcing a cache fault. Similarly, the test problem being measured may call procedures or make data references which force cache faults. In a best case, one pass through the timing loop code and test problem will load all the instructions into the cache where they will remain for subsequent iterations of the timing loop, permitting all instruction fetches to be made from the cache with no main memory accesses. The time to execute the timing loop instructions may sometimes include cache faults and sometimes avoid cache faults, depending on the contents of the instructions enclosed by the timing loop and the memory alignment of the timing loop code (with respect to their mapping into cache frames). On a large target system, there may be sufficient cache memory to hold an entire test problem (or even the entire test program it is a part of). When the ACEC testing is performed on an unloaded system and typical applications normally execute in a multi-programming environment, the ACEC test results may overestimate the speed achievable by the processor.

Designing programs to minimize cache faulting behavior is rather difficult. It is typical practice to assume that the cache fault rate will be roughly constant. It also helps analysis that the time penalty for a cache fault is much smaller than for a paging fault in a virtual memory system, making the payoff from adapting code to minimize cache faults much less than for paging systems. A simple way to exploit a cache memory system is to write small programs, which fit entirely within the high speed cache. The ACEC includes both large and small test problems.

- Analyzing the performance of processors which prefetch instructions is more complex than for processors which do not. When an instruction is ready for execution (prefetched) as soon as the flow of control is ready to execute it, the processor will execute it faster. This effect can make the time to execute a linear sequence of statements appear faster than expected. For example, to execute the statements "X1 := Y1; X2 := Y2;" can take less than twice the time to execute as "X1 := Y1;" This effect is explicitly examined in the ACEC in test problems SS11, SS635, and SS634 which contain one, two, and three integer assignment statements (respectively) using distinct variables. If the execution times for these statements form a linear series, there is not a "prefetch" effect.

Instruction prefetching can make the execution time for the instructions at the beginning and end of the timing loop vary depending on the code being en-

closed. If the flow of control sequentially "falls" into the instructions forming the end of the timing loop, the instructions will have been prefetched and the processor will not have to wait before executing the instructions. On the other hand, if the flow of control has jumped to the instructions at the end of the timing loop, the instructions may not have been prefetched and the processor may have to wait for them. On some prefetching processors, the outcome of a branch may be predicted and instructions fetched. This is straightforward (although requiring more expensive hardware) for unconditional branch and branch-and-link instructions, and branch prediction logic permits some processors to anticipate the outcome of branches and prefetch instructions. For example, when the last statement in the test problem is a procedure call resulting in the last instruction executed in the test problem being a RETURN, or when the test problem terminates with something like:

```
      IF boolean_function_returning_true THEN
          proc0;
      ELSE
          RAISE program_error;
      END IF ;
```

where the statement will take the THEN alternative and then execute a machine level jump instruction to get to the end of the IF statement. Prefetching could cause the time to execute the NULL loop to be different than the time to execute the timing loop code in a test problem.

The ACEC minimizes the potential for systematic errors due to prefetching by including a subprogram call within the timing loop (in STOPTIME0) which will force a break in the prefetching and preclude prefetching of the decrement and test code at the end of the timing loop.

Because the NULL loop sequentially "falls into" the end of the timing loop, subtracting off this NULL loop time may add a systematic error (of one prefetch setup time) for test problems which branch into the end of the timing loop code on prefetching processors. Some prefetching processors include logic which will anticipate branching instructions and will fetch the instructions beginning a subprogram, and some will even prefetch both alternatives of a conditional branch. On such processors, the coding of the timing loop will not suppress prefetching, but the actual measurements returned will be accurate since the instructions will be prefetched anyway. The specific conditions under which any particular processor prefetches instructions can be very complex. For complex processors, the results of short test problems should not be given too much emphasis, although they will still be useful for code expansion

measurements and for some comparisons to related problems.

- Memory bank contention can introduce delays. High speed memory systems are designed to process several memory requests concurrently. This supports I/O devices with Direct Memory Access (DMA) connections. High performance CPUs also exploit concurrent memory access by having several memory requests outstanding simultaneously — for example, prefetching the next instruction and making a data reference. Such memory systems do not work at full speed when requests to the same bank of memory are made. Although true multi-port memory chips exist, they are *much* more expensive than standard memory chips and are not ordinarily used for main memory systems.

On multi-bank systems, the alignment of instruction and data references can introduce delays in completing memory references.

Many simple processors do not have a memory bank organization, and this complication in performance prediction does not occur.

- To save costs, a memory system may contain memory of various speeds. Code executed in slow memory will not run as fast as when it is loaded into higher speed memory.

The ACEC approach to this potential problem is to acknowledge that it may exist. ACEC users need to be aware of the characteristics of the processors they are testing. If the NULL timing loop is executed out of high speed memory and some test problem is executed out of slow speed memory, then the measurements will contain an error term reflecting the difference in execution time between executing the NULL loop in different speed memories. The ACEC MEDIAN tool will detect this difference for small test problems because they will be reported as faster than expected (perhaps zero time) when the NULL loop is executed from slow memory and the test problem is in fast memory, or they will be reported as much slower than expected when the assignments to slow and fast memory are reversed.

- Alignment in memory can impact performance, and different alignments of the timing loop code in the NULL loop and in each test problem can introduce systematic measurement errors on some processors. For example, some memory systems respond with fixed width aligned blocks, such as 64 bit wide chunks which always have a beginning byte address which is $0 \bmod 8$. A memory request which is not aligned with the memory system will take multiple memory cycles to service. For example, fetching a 64 bit word which is not aligned on a $0 \bmod 8$ byte boundary will be slower than when it is aligned. When the instructions forming the timing loop have different memory alignments, then a systematic error will be introduced.

The alignment effect discussed here is relative to instructions rather than data references, since the references to the variables which control the timing loop in the NULL loop and in general test problems will be to the same library scope variables, whose alignment will not change.

The ACEC approach to this problem is to test as part of INITTIME that the NULL loop has reliable times when replicated four times. It is hoped that the timing loop will not have the same alignment by chance. Because each target processor may be sensitive to different alignments, it is not possible to *systematically explore all variations which might be important to some target processor*. The variations between four null loops are indicative of the variations which may occur in the code, and these are printed by INITTIME. The variation between measurements of the null loop time is printed (differences between test problem measurements less than this value should not be considered significant) and this variation is used to calculate the fastest execution time where a variation of this magnitude will be less than the requested error tolerance. The variation in null loop measurements is also used to detect probable faults in measurements. If the elapsed time to execute a test problem is less than for the null loop, a negative value would be calculated for the test problem execution time — if the estimated negative execution time is less than the variation in the null loop, the result is reported as a zero; if it is greater than the variation in the null loop, it is reported as an unreliable measurement and the user is advised to disregard the measurement. If the initialization routines do not produce consistent times, the ACEC user will be told that there is a problem, and may have to accept the presence of some failures to achieve consistent measurements. Users may have to accept that when the NULL loop measurement is not consistently repeatable, all calculated timing loop measurements reflect the error associated with the computation of the NULL loop time (as displayed each time the test program is executed). The relative error introduced by considering this effect can be large for short test problems.

Systematic errors in the execution time measurements can result when the timing loop executes differently in the null loop than for general test problems. This source of potential errors has been minimized in the ACEC by careful design and testing of the timing loop code. For some targets, the results may not be completely successful. It is hoped that this discussion will alert ACEC users about the potential of systematic errors in the ACEC measurements they obtain. The ACEC includes related test problems to detect the presence of systematic errors not compensated for.

The fact that a test problem can take a different amount of execution time depending on the memory address it is loaded into, or the nature of the Ada statements preceding it is a direct result of the memory organization of some processors. This fact complicates the interpretation of results and essentially precludes the naive extrapolation of short test problems to overall system evaluations.

The ACEC approach of including both small and large test problems will help in making fair assessments.

5. Pipeline processors exaggerate the effect of statement context on performance. A pipeline processor can have several instructions at different stages of execution at any one time: fetching the instruction; performing the address computation; fetching the data operands; performing address computations; performing arithmetic/logical operation; and storing results. The time to execute any particular instruction will vary depending on the state of prior instructions in the pipeline. For example, when one instruction references the result of a prior instruction, the processor must wait until the result is available. This condition is the definition of a pipeline interlock. Some current designs of pipeline processors have no provisions for hardware interlocks and require the programmer (that is, the high level language compilers) to insure that the executed code is hazard free.

The performance of a pipeline processor may be sensitive to the mix of operations performed. There may be several dedicated functional units, as in machines derived from the CDC 6600 series processors. Because an instruction will have to wait for an available logical processing unit, performing a sequence DIVIDE, DIVIDE, ADD, ADD may be much slower than the sequence DIVIDE, ADD, ADD, DIVIDE (assuming that data dependencies between the instructions permit the reorganization). Short test problems, which execute only a few instructions before they branch, will underestimate the performance benefit of pipeline processors, since they present the pipeline with worst case behavior. The timing loop will "break" the pipeline because it contains an external procedure call.

Long test problems, which execute a long sequence of instructions before branching, will execute well on pipeline processors.

On a pipeline processor with a "loop mode," after the first time the loop is executed, the CPU will retrieve instructions from the pipeline and will not access memory to fetch instructions until the loop is exited. This can improve the execution time of many loops dramatically, since high speed CPUs tend to be limited by the speed of memory references. A statement like "sum:=sum+x(i);" can be repeatedly timed to take "n" microseconds, but if it is the only statement in the body of a loop, it might well take much less than the time for a null loop plus "n" times the number of iterations of the loop.

A pipeline processor presents potential problems to programmers constructing benchmarks, to readers of benchmark results, and to programmers attempting to obtain optimal performance on the processor. They make the execution time of a sequence of machine instructions depend strongly on the surrounding instructions. This fact must be remembered when considering the reported result from any test problem. For short test problems, the pipeline will never be fully "primed" and the reported result will be a pessimistic estimate of performance.

Linearly extrapolating from the speed of a simple statement measured as a test problem to executing the same statement within a tight loop can dramatically underestimate the speed of a pipeline processor. When a tight loop contains an external procedure call (not suitable for inline expansion), performing the CALL (or Branch And Link, or equivalent) instruction and returning control will force two "breaks" in sequential instruction prefetching, which would not occur if the test problem did not include any branching instructions. Such a call occurs in the ACEC timing loop code. When the ACEC measures the time to execute one simple assignment statement, it includes the time to "prime" the pipeline. A test problem for two or three simple assignments in a row may take much less than two or three times the execution time of a single assignment. When a test problem contains a loop, the code for the test problem might be small enough to fit within the "loop mode" of the pipeline processor and require no references to main memory for instruction references. Such a "loop mode" could run much faster than a simple linear extrapolation would predict.

When examining measurements on pipeline processors with loop modes, small test problems containing simple loops should run much faster than problems without loops. This is an expected consequence of the target machine. Users must understand this aspect of the target machine when they use the ACEC test results to extrapolate the performance on other programs. The vendors of such machines typically include advice on how to best exploit such target machines. General advice with respect to the machine properties should be verified with the Ada compiler being used, since the compiler may introduce overheads which preclude exploiting the features of the hardware in Ada in the same way that it can be exploited in other languages. It might perform all array aggregate operations with calls on routines in the RTS, which preclude using loop mode operations, whereas an explicit FOR loop structure may result in much faster execution.

The ACEC has attempted to minimize systematic measurement errors in the timing loop by using a measurement process which will work for pipeline and nonpipeline processors. For the individual test problems, the effects of pipeline processors should not be viewed as a source of systematic errors which must be minimized, but as an

unavoidable aspect of some target processors which application programmers will have to deal with. It is not simply a "feature" which hardware designers put into processors to complicate the life of benchmark developers. The ACEC will execute the test problems as written on each processor it is executed on. It does not claim to be optimal code for each target processor.

The idea that each language feature will have a fixed execution time and that a user can estimate the total time for a procedure to execute by adding up the time for each feature multiplied by the number of executions of the statement is oversimplified. Optimizing compilers falsify this assumption on simple processors — moving operations out of loop, removing dead assignments, evaluating foldable expressions at compile time, reusing common subexpressions, and applying all the techniques available to it to improve generated code. Pipeline processors falsify the assumption on any sequence of machine instructions. Prefetching processors, to a lesser extent than pipeline ones, also can falsify the assumption for arbitrary sequences of machine instructions.

The ACEC test suite contains both large and small test problems. For some target processors, it will be unsafe to extrapolate from short test problems.

How fast a pipeline processor executes a particular application is influenced by how much branching (either due to subprogram calling or to conditional statements) the application does and by the patterns of memory references. The results from the ACEC test problems relative to loop unrolling are of particular interest on pipeline processors, since this is an optimization technique which can make a large difference. If unrolling loops prevents a "loop mode" operation, it can degrade performance; otherwise it might enhance performance considerably. Loop unrolling is the expansion at compile time of a loop into the equivalent sequence of statements.

The ACEC approach to minimizing systematic errors due to execution on pipeline processors relates to insuring that the execution time of the timing loop code in the NULL loop will be comparable to the execution time for general test problems. Including an external procedure call in the timing loop will introduce a "break" in the pipeline.

6. On a multiprogramming target system, concurrent users will contend for resources. Not all the time between the beginning and end of a test problem is necessarily spent executing that problem on a multiprocessor. The processor may have been diverted to execute another, higher priority task. Even if a user arranges for highest priority processing for execution, system services may pre-empt system resources — for example, servicing I/O completion interrupts is usually a higher priority operation than any user program.

There are two things an ACEC user can do to minimize the impact of contending users. First, the user can arrange for exclusive use. This is not typical of usage of multiprogramming systems, but the primary audience for the ACEC is MCCR applications, many of which are executed in a stand-alone environment. A small amount of contention can be compensated for by the design of the timing loop, which uses multiple cycles, and by the use of minimum time for analysis. FORMAT will extract the minimum time among the repetitions of the test problem for analysis by MEDIAN. If contention occurs in short bursts, the effects will be limited to the cycle where the burst occurred and it will always *add* time to the measurement. Because the timing loop synchronizes with the system clock, if the interference from contending tasks stops before the next system clock tick, it will not contaminate the next measurement cycle. Taking the minimum measurement for timing analysis is comparable to assuming that the minimum measurement does not have spurious additions.

Second, the user may adapt the ACEC to use CPU time measurements. For most problems, the difference between CPU time and elapsed time on a dedicated processor should be small. Exceptions include:

(a) Tasking tests.

On multiprocessors, parallel processing can produce smaller elapsed time than CPU time. On both uniprocessor and multiprocessors, the accounting system which "charges" time to processes may treat task scheduling as an overhead operation.

(b) I/O test problems. The CPU may be idle, waiting for the completion of an I/O operation.

Here, CPU and elapsed time are not comparable. In general, CPU time measurements require the use of system dependent routines, which are not universally available. Most multiprogramming operating systems which maintain CPU time measurements do so for accounting purposes and the accuracy demanded for this task is not great. Few systems accurately allocate the CPU time during interrupt processing to the job which was responsible for the interrupt. The additional overhead required to proportion this time correctly is substantial, the differences in billing would be very small, and it is not always clear what job should be charged (consider servicing the system clock).

The approach of the ACEC to minimize this source of systematic errors is to instruct users to try to execute the test problems when there are not concurrent users on the system, or to use CPU time measurements. The timing loop code contains checks for consistency of measurements and will indicate failure to converge. When contending users are executing on the target, the measurements will often be flagged

as unreliable.

7. The system clock may not be accurate. There are tests for clock accuracy in the test suite. The TESTCAL1 and TESTCAL2 programs test for long term drift in the clock. If these programs revealed inaccuracies, the ACEC users should have the problem corrected before proceeding. The timing loop initialization code, INIT-TIME, checks for jitter (random variation in the clock) and compensates for it by ensuring that each test problem will execute for a long enough time that the relative error attributable to jitter will be less than the tolerable relative error.
8. The code for similar looking constructions may vary if certain "magic numbers" are crossed. For example, on a DEC VAX, the length of an instruction can vary, depending on how many bits are required to reference a data object, or to encode a literal value. Long format instructions can be slower than short format instructions. If relative jump instructions are used with a variable length operand, the jump instruction to return to the head of the timing loop can vary depending on the size of the test problem. Small test problems, including the null loop, may be able to use short format instructions, while larger test problems may require longer formats. Therefore, large test problems will be incorrectly reported as slower by the difference between the execution times of jump formats.

The ACEC approach to this potential problem is basically to hope that the systematic errors introduced when the timing loop overhead for NULL loop is different from the overhead of the timing loop for a general test problem which is large enough to force a long format branch will be small relative to the requested error tolerances.

In the test problems themselves, compilation systems will be presented with specific code sequences, and the compilers will either generate "short format" instructions or not. Comparing times of related test problems can reveal the presence of special formats.

9. The size of the test program can modify performance. Some compilers do not optimize large programs — some analysis might take excessive times on large programs and are not attempted, resulting in the same expressions producing better code in small blocks than in large blocks. This might result in the code generated for the timing loop of the null loop being optimized (since it usually occurs early in the source text of the programs) while the timing loop for the test problem being measured is not optimized. This can introduce a systematic error into the measurements.

ACEC test problems SS769 through SS773 all test the same statement for consistency. If the performance of a system degraded on large programs, examination

of these results might disclose this fact, although it is quite possible that some compilers will have "cutoffs" which are larger than will be detected by these test problems.

It is possible to detect the presence of various sources of systematic errors by including in the test suite example problems which are variations on the same construction which give the Ada system being tested the chance to display different performance under conditions designed to expose the presence of systematic errors.

One way is to present test problems which repeat the same construction one, two, and several times. If the measured times are not multiples of the single occurrence, there is a measurement anomaly present. Discrepancies might be due to cache organization or memory alignment. They might be due to different code being generated for the timing loop itself. They might be due to possible sharing of setup code between repetitions of the constructions. They might be due to better flow through prefetching. The detailed explanation for such discrepancies will require examination of the machine code and the target hardware. While the ACEC does not perform this search automatically, the presence of timing anomalies can be detected by comparing the measurements between sets of related test problems. For example:

- SS36 and SS642. These test problems are, respectively, one procedure call and ten calls on the same procedure.
- SS11, SS635, SS634. These test problems are, respectively, one, two, and three simple assignment statements (using different variables). The Single System Analysis (SSA) looks at these results in "Sequence of Assignments" in the Language Feature section of the main report.

It would be simpler for benchmark developers if processors were simpler to analyze, but the ultimate users of compilers derive considerable benefits from the non-simple aspects of processors, and it is the responsibility of benchmark developers to develop a capability to evaluate real processors.

5.3 HOW TEST PROBLEMS ARE MEASURED

Each test problem is measured by "plugging it into" a template which will, when executed, measure and report on the execution time and size of the test problem contained within it.

The timing loop code consists of four (4) code files which are incorporated into the source by a preprocessor (INCLUDE) which supports text inclusion. The body "INITTIME" is

included once per program and contains code to initialize the timing loop variables (as discussed later in this section). The other code files bracket each test problem to be timed and provide a place for writing an identification of the test. They are responsible for computing the execution time and code expansion size of the test problem they enclose, and for outputting the measurements obtained. The general form for all test programs is:

with global; use global;	
with calendar; use calendar;	
	-- declarations
begin	
pragma include ("inittime");	-- initializations can go here
	-- once per program
	-- initializations can go here too
pragma include ("starttime");	-- first test problem
...	-- test problem code goes here
pragma include ("stoptime0");	
put("...");	-- name and description goes here
pragma include ("stoptime2");	
pragma include ("starttime");	-- second test problem
...	-- test problem code goes here
pragma include ("stoptime0");	
put("...");	-- name and description goes here
pragma include ("stoptime2");	
...	-- additional test problems enclosed by
...	-- starttime / stoptime0 / stoptime2 follow

Figure 7: Timing Loop Template

In Figure 7 the "..." after the `PRAGMA INCLUDE("inittime")` would be replaced by initialization code, that after the `PRAGMA INCLUDE("starttime")` would be replaced by the test problem to be timed, and that after the `PRAGMA INCLUDE("stoptime0")` would be replaced by a put (or a sequence of `PUT_LINE` and `PUT` statements) which print the problem name and an English description of the problem. It is appropriate to include lists of related tests and the purpose of the test.

The purpose of each of the included files is as follows:

- `INITTIME`. This code initializes timing loop variables and computes the execution time and code expansion space for the null loop. It is executed once per program.
- `STARTIME`. This code is the head of the timing loop.
- `STOPTIME0`. This code is the tail of the timing loop.
- `STOPTIME2`. This code outputs the measurements collected on the current test problem.

5.4 TIMING TEST PROBLEMS

On most target systems, the resolution of the system clock is such that many instructions can be executed between changes in the clock. This observation leads to two conclusions.

First, the test problems need to be iteratively executed to insure that they will consume sufficient elapsed time to permit reliable measurements to be made. This is a standard benchmarking technique. It involves dividing the elapsed time measurement by the number of iterations and subtracting off a precomputed null loop time.

Second, where the system clock is accurately (but coarsely) maintained, a software vernier can be constructed which permits high resolution measurements to be made. Any measurement can be no more accurate than its standard. However, the accuracy of the standard is not the same as the precision measurement process which compares measurements to the standard. On many IBM S/360 computer systems, the time measurement standard is based on the 60 Hz power line frequency. This is considerably more accurate over a small interval than the 16.7 millisecond resolution of the clock would indicate, but not as accurate as the 26 microsecond resolution of the value returned by a supervisor call requesting the time.

Accurate measurements can be performed whenever there is an accurate hardware clock with minimal jitter and a short program loop whose time is accurately known that can detect when the clock is incremented.

5.4.1 Clock Vernier

The timing loop code proceeds as outlined in Figure 8, where the major vertical lines indicate a time when the clock changes value: that is, a "clock tick."

1. Synchronize the beginning measurement with the system clock by looping until the clock changes. This is shown at time S1 of Figure 8.
2. Execute the test problem. This is shown at time S2 of Figure 8.
3. Loop, reading the clock until it changes, counting the number of iterations this takes. This is shown at S3 of Figure 8.

The time measurement desired is then $S2 - S1$. This can be computed as $S3 - S1 - \text{NUMBER_OF_ITERATIONS_COUNTED} * \text{TIME_PER_TICK}$. Where `time_per_tick` was computed during program initialization.

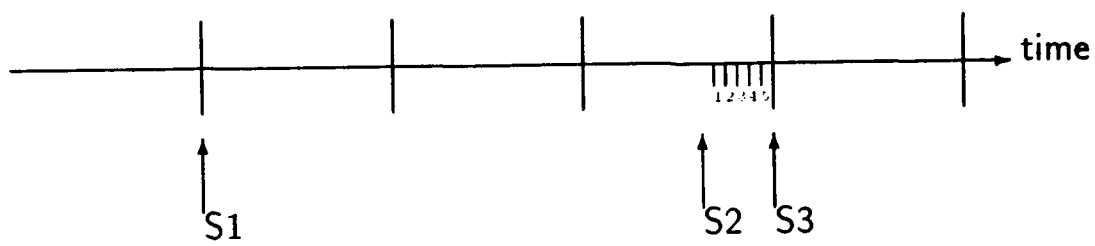


Figure 8: Clock Vernier

The principles of a software clock vernier have been published before by both Wichmann (ALGOL60 Compilation and Assessment, 1973) and Koster (Low Level Self-Measurement in Computers, 1969).

Some systems overlap the display of output results and the continuation of normal processing. On these systems, when output is completed, an interrupt will be generated which must be serviced to complete all the output operations. In addition, while output is in progress, it will steal memory cycles. To insure valid results on systems which do this, it is important to insure that processing associated with output operations is completed *before* starting the next measurement. This potential source of systematic errors is minimized by including a one second **DELAY**, sufficient to permit I/O completion, before starting each measurement. This is done in **STARTIME**.

The time between ticks of the system clock is typically many instruction execution times. A one millisecond time between clock ticks is a thousand instructions on a one Million Instruction Per second (MIP) machine. To obtain reliable estimates of test problem execution times, the ACEC timing loop executes each test problem repetitively within a loop, and subtracts out an estimate of the null loop overhead which it computes as part of the initialization of each test program. The number of iterations to perform each test problem is dynamically determined. This permits execution on different target processors of different speeds, without having to modify the source text or interrogate an operator interactively to request a proper value for the *iteration count*. While the precise value of the iteration count is not critical, too large an iteration count will take excessive computer resources and elapsed time to collect measurements, while too small a count can produce inaccurate measurements.

On some systems the granularity of the clock will be small enough and/or the time to execute the call on the function **CALENDAR.CLOCK** will be large enough that *time_per_tick* will be greater than the clock granularity. On these systems, the clock will always return a different value each time it is read and the vernier compensation will be pointless. On such target systems, the use of the software vernier will not add precision to the measurement process and will actually introduce a small error term into the measurements since the timing loop will have to execute the vernier loop compensation code (basically adding the time to perform a call on **CALENDAR.CLOCK** and a comparison of two values of type **TIME**) which will be added into the null loop compensation time and subtracted out of each measurement. On such systems, the timing loop code could be simplified by removing the clock vernier loop. The ACEC does not distribute a version of the timing loop code without a software vernier because:

1. Using a software vernier on target systems where time per tick is greater than the clock granularity does little harm.

2. Not using a software vernier on target systems where `time_per_tick` is greater than the clock granularity will make the measurements take longer by requiring more iterations to obtain consistent measurements than otherwise necessary. It will result in accepting less precise measurement results.
3. Most target systems explored during the development of the ACEC have a `time_per_tick` which is smaller than the clock granularity. Using a software vernier is usually beneficial.
4. The use of the ACEC would be complicated if it required each user to select variations of the timing loop based on the relationship between `time_per_tick` and the clock granularity. Interested users could make this system adaptation if they desire.

The remainder of this section describes the details of the timing loop code. This is provided here for reference, so that an ACEC user can understand what the number printed after executing an ACEC test program means (SIZE, MIN, MEAN, INNER LOOP COUNT, OUTER LOOP COUNT, SIGMA, and the indicator field). An ACEC user not concerned about the meaning of the count fields can skip the rest of this section.

The timing loop code requires some initialization before it can be used to measure test problems. The null loop time is computed in the initialization code, as is the value of "time per tick." The jitter compensation time calculation proceeds in two steps. The initialization code repetitively synchronizes with the system clock and then counts the number of times it can read the system clock until the clock changes. If no jitter exists in the system clock, this count will be constant (barring quantization errors). The standard deviation of the count of ticks is a measure of the system jitter. The minimum value of the elapsed time for a test problem measurement is then the maximum of:

1. The standard deviation compensation. A minimum elapsed time is computed such that the observed standard deviation is less than the requested error tolerance with the requested confidence level.
2. The quantization compensation. A minimum elapsed time is computed such that an error of one time per tick interval will be less than the requested error tolerance with the requested confidence level.

The ACEC timing loop code consists of two basic phases. The first phase is a search for the appropriate number of times to iterate the test problem to produce a reliable measurement. The second phase repetitively executes the test problem (using the iteration count from the first phase) until consistent measurements are obtained. There are refinements added to these basic phases to enhance accuracy and prevent any one test problem from executing for an "excessive" time.

The code for the ACEC timing process contains two loops, an inner loop and an outer loop.

The inner code loop, as outlined in the clock vernier discussion, will:

1. Synchronize with the system clock and then;
2. Execute the test problem `GLOBAL.NCOUNT` times — `NCOUNT` is an integer variable whose value is controlled by the outer code loop and then;
3. Re-synchronize with the system clock, counting loops as discussed in the vernier compensation discussion and then;
4. Compute an estimate for the test problem execution time, based on null loop time and the vernier compensation term.

The outer code loop implements the two logical phases by executing and examining the results of the inner loop.

The first phase of the timing process is a search for the appropriate value of `NCOUNT`. That is, the smallest value of `NCOUNT` large enough that the measurement will provide a reliable estimate. If the current value of `NCOUNT` is too small, it is replaced by $2 \times \text{NCOUNT} + 1$ and the inner loop executed again. The search starts with `NCOUNT` having a value of one. `NCOUNT` is never made larger than `BASIC.ITERATION.COUNT`. Without this limitation, for test problems translated into nulls, an infinite `NCOUNT` would not be large enough to result in a reliable measure of problem time and the phase 1 timing loop logic would try to increase `NCOUNT` without limit — it might eventually stop when it exceeded `SYSTEM.MAX INT` (but when constraint checking is suppressed that is not guaranteed) but this could represent more elapsed time than either desirable or necessary to determine that a test problem has been translated into a null. The sequence of values of `NCOUNT` is 1, 3, 7, ... $2^{**}N-1$. This sequence makes it possible for `NCOUNT` to assume the value $2^{**}15-1$ without risking integer overflow on machines which only support sixteen bit integers.

The second phase basically uses the value of `NCOUNT` from the first phase to perform the inner loop code repetitively, testing after each cycle if it should terminate the measurement of the test problem. Cycles are repeated until:

- The number of cycles is greater than `GLOBAL.MAX.ITERATION.COUNT`.
`GLOBAL.MAX.ITERATION.COUNT` is a named number which limits the number of cycles the timing loop will perform. It can be modified by an interested ACEC user. It should not be made smaller than `MIN ITERATION COUNT`. If this value is made large enough, it is likely that most problems will never terminate due to exceeding it — the confidence level test for termination is based on the standard

error in the means which is defined as the standard deviation of the samples divided by the square root of the number of samples: when the measurements are statistically independent, the standard error will asymptotically approach zero (it is *not* necessary to make any assumptions about the distribution of the errors, such as being normally distributed).

- The number of cycles is greater than GLOBAL.MIN ITERATION COUNT and a Student's t-test indicates that, to the confidence level requested in GLOBAL, the timing measurements are being drawn from a sample with the same mean.

Refer to standard statistical text for a detailed description of the Student's t-test, such as Introduction To the Theory of Statistics, by A. Mood and F. Graybill. The confidence level is built into GLOBAL. It may be modified by adjusting the named number GLOBAL.TIMER CONFIDENCE LEVEL and the contents of the array GLOBAL.T VALUE, as described in the comments in the source file GLOBAL.ADA.

- When any individual test problem has executed for more than thirty minutes the test problem is terminated when the outer loop code gets control and a descriptive error message is generated. This test is intended to prevent excessive execution times and should not occur often. An informational message is also written when more than five minutes have passed since a test problem started and the outer code loop gets control.
- To prevent any one test problem from taking excessive elapsed time when measuring CPU time, the outer timing loop terminates when the CPU time is less than 10% of the elapsed time and the elapsed time is greater than ten seconds. Without this modification, the timing loop code would execute for days in measuring the CPU time for a "DELAY 1.0;" statement. This modification only affects test problems which are not CPU bound: language features whose use can make a test problem non-CPU bound include DELAY statements; I/O statements where the CPU will be idle until a physical I/O operation completes; perhaps tasking on some target systems where task scheduling is charged to system overhead rather than to each user job; or a system which has heavy contention from other (higher priority) user jobs or system daemons. If this condition is detected, it will be noted on the output and an "unreliable measurement" error code reported.
- The outer code loop may be "terminated" when large variations are observed between cycles and Phase 2 restarted with a larger value of NCOUNT. This "restarted" Phase 2 will then be terminated by one of the conditions listed here. Large variations between cycles can be caused by transient conditions, such as when the initial execution of a test problem on a virtual memory system causes a page fault forcing a wait until the code is loaded, or when a system daemon usurps the CPU (to process

an interrupt or to provide periodic service to a timed queue). It is best to avoid taking measurements on a system while it is processing transients. If the timing measurement will converge it would be better to increase the value of NCOUNT and reenter Phase 1. If the transients go away and the time measurements stabilize, then a reliable measurement will be generated. If the behavior does not stabilize, the system should "waste" little execution time in an attempt to produce good measurements. When a test problem takes a variable amount of time for each execution, increasing the number of iterations and dividing by the iteration count will not always converge. Increasing NCOUNT in such a case will simply make the ACEC run longer (perhaps *much* longer) trying to measure the test problem. Therefore, the ACEC timing loop code only increases NCOUNT in Phase 2 when the current measurement satisfied all the points listed below:

1. It is greater than twice or less than half the minimum time.
2. It is less than $1.0 / \text{real}(\text{NCOUNT})$.
 This (relatively) arbitrary cutoff is perhaps more intuitive when written as "if time * real(NCOUNT) > 1.0 then ... do not increase NCOUNT ... end if;".
 Time * NCOUNT is the total time taken to execute the test problem. When this total time is greater than one(1) second the time is large enough to produce reliable timing estimates. Estimates based on small values of total problem time will be sensitive to small errors in clock measurements.
3. NCOUNT is less than BASIC ITERATION COUNT.

An informational message is displayed when the variation in measurements between cycles is more than a factor of two, but the cutoff parameters prevent extending the Phase 1 search.

This "termination" and reentry into Phase 1 serves two purposes. First, using this technique, the ACEC timing loop can produce more accurate measurements by ignoring transient behavior. Second, by not assuming that *all* variations are due to transient noise, it will not spend an inordinate amount of time increasing the iteration count to force a test problem timing measurement to converge when the test problem is inherently variable.

During initial development of the ACEC timing loop, the code always reentered the Phase 1 search whenever the times varied by a factor of two. On one trial system, some test problems containing I/O operations executed for over a day without terminating because it always reentered Phase 1 and increased NCOUNT — I/O performance on this system varied more than the confidence levels were set for.

5.5 CODE EXPANSION MEASUREMENT

The measurement of code expansion is performed in the timing loop. This can be done in either of two ways — using label'ADDRESS attributes or a GETADR function. In both cases, the measurement is computed as the difference between the addresses at the end of "STARTIME" and the beginning of "STOPTIME0". These techniques compute the differences between values of type SYSTEM.ADDRESS. The resulting sizes can be multiplied by STORAGE_UNIT to get the code size in bits. Presenting size in bits will permit easy comparison between target machines with different values of word sizes (values of STORAGE_UNIT). Details of measurements are discussed in the User's Guide, Section "CODE EXPANSION MEASUREMENT".

Out of line code which supports a construction used in a test problem will not be counted in the code expansion measurement. RunTime System (RTS) routines will not be measured here. If the compiler calls on runtime library routines to translate a feature, the space measured will be that required to perform the call. This differs from timing measurements which will include the time to perform any called subprogram.

5.6 CORRECTNESS OF TEST PROBLEMS

There are several issues associated with correctness, which will be discussed in turn along with the steps taken in the ACEC to minimize problems due to them. Some of the issues have been mentioned in other contexts in this document, but deserve explicit discussion. Issues include:

- Validity of test problems;
- Correct translation of test problems;
- Satisfaction of intention;
- A priori error bounds; (see Section 5.2).
- random errors; (see Section 5.2.1).
- and systematic errors.(see Section 5.2.2).

5.6.1 Validity of Test Problems

Considerable care has been taken to insure that the test problems are valid Ada. The test problems will have been attempted on five different Ada compilation systems — the cases where the problems have failed to compile or execute are noted in the Version Description Document (VDD) Appendix IV, "QUARANTINED TEST PROBLEMS".

Some test problems are system dependent. Any test suite which seeks to address all the Ada features of interest to MCCR applications must include tests of system dependent features. These test problems are noted in the VDD Appendix VII, "SYSTEM DEPENDENT TEST PROBLEMS".

5.6.2 Correct Translation of Test Problems

Determining the correctness of an Ada implementation is not the charter of the ACEC. That is the responsibility of the ACVC project. However, it is misleading and undesirable to credit an implementation for producing wrong answers quickly. Self-checking is included on some test problems where nonobvious errors occurred during testing. If a test problem provokes a compile time error or aborts during execution, failure is obvious and does not call for self-testing code. Users who detect errors in an Ada implementation when executing the ACEC should report them to the compilation system maintainers for correction.

If a difficulty is discovered with a test problem on one particular system, this may be sufficient to justify changes to the test suite, depending on the nature of the difficulty encountered and its impact on portability. The appropriate response to the discovery of errors in a compiler may be to simply report them to implementors for correction (and perhaps submit a copy to the ACVC) if the error seems to be one which might be generally encountered.

Any test problem which failed on one or more of the five trial systems will be "quarantined." A listing of the failed problems will be reported in the VDD Appendix IV, "QUARANTINED TEST PROBLEMS". This appendix will list for each quarantined test problem a count of the number of systems on which it failed. Users can refer to this information in the VDD to quickly discover if a problem they are having difficulty in running has uncovered problems in other systems also. It may be that the system they are testing shares limitations with a previously tested system.

5.6.3 Satisfaction of Intent

Test problems are included in the test suite for specific reasons. They may fail to satisfy their intentions for two reasons.

- A problem might be optimized with respect to the timing loop.
- Specific optimization test problems might be susceptible to an unanticipated optimization, resulting in the problem not being a test for the original optimization.

For the timing loop code to work it is necessary that the test problem be executed once for each iteration of the timing loop. It must follow the same computational path on each iteration, or it will not be valid to divide by the number of iterations to compute the individual problem execution time. For measurements on the test problems to realistically represent the behavior of the test problem in a general context, it is necessary to insure that code is not optimizable with respect to the timing loop. In particular, the test problems should:

1. Not be loop invariant, in whole or in part. If the code fragment is completely invariant, one execution of it would suffice for all values of the loop iteration count. Even where the code fragment is not completely invariant, it might contain a (sub)expression which is invariant and subject to loop motion — such as “a**2” where “a” is a loop invariant variable.
2. Use live variables. Variables containing the results of a computational sequence in a code fragment must be referenced after being assigned. If they are not, an optimizing compiler might determine that the computations are irrelevant and ignore them.
3. Not be strength reducible with respect to the timing loop.
4. Not be unduly foldable. A code fragment which is interesting to measure often *requires initialization code to insure correct and repeatable execution*. Any such setup code should not permit the “interesting” code to be evaluated at compile time.

When the flow through a code fragment is determined by the values of variables that the code later modifies, for example,

```
IF first_time_flag = false THEN
    initialize_procedure;
    first_time_flag := true;
END IF;
...
```

it is necessary to insure that the same path is followed on each repetition. If the test problem is following different computational paths on each repetition of the timing loop, it is incorrect to compute an average time by simply dividing by the number of repetitions.

Some code fragments will, when executed repetitively, generate a numeric overflow, which will raise an exception and disrupt the flow of control. Consider the statement

```
i := i + 1 ;
```

where "i" is never reset. This will eventually reach the integer overflow limit and raise an exception (if checking is not suppressed). Code fragments with either of these characteristics should be modified. Numeric variables should have the same value at the start of each iteration, either by reinitializing them or by insuring that their new computed value equals their initial value.

If a user determines that a problem does not satisfy its intent on some target, that is sufficient reason to report a problem. If a compiler is able to find a better way to translate a test problem, that is not a fault of the compiler. It is not necessarily a reason to withdraw the test problem — the performance of various systems on this example may be a good way to expose the use of the unanticipated optimization techniques. If such an optimizing compiler results in the test suite not having any remaining tests which address the originally intended optimization, it may then be desirable to construct and incorporate into the ACEC test suite additional test problems which are not amenable to the unanticipated optimization — and update the comment block of the original test problem to reflect the optimizations and the new related test problem.

6 STATISTICAL BACKGROUND FOR MEDIAN

This section of the guide presents the statistical background for the analysis program, MEDIAN. Readers without a statistical background may wish to skim this section. All readers should read at least the first few pages.

MEDIAN assumes a statistical model and fits it to the observed data. It also checks on how well the model describes the data. The general model, and its application to timing data, code expansion data, and RTS size are each discussed in turn.

6.1 MATHEMATICAL MODEL

MEDIAN assumes that the time (or space) that a system takes to execute a problem can be approximated as the product of two factors: one depending only on the system and one depending only on the problem.

$$\text{Time (System , Problem)} \sim \\ \text{System_Factor (System)} * \text{Problem_Factor (Problem)}$$

MEDIAN computes values for System_Factor and Problem_Factor to best fit the data. It scales the factors so that the System_Factor for the system first named in the enumeration type defining the values of type "systems" is 1.0.

$$\text{Time (Systems , Problem)} = \\ \text{System_Factor(Systems)} * \text{Problem_Factor(Problem)} \\ * \text{Residual(Systems,Problem)}$$

This multiplicative model is intuitively appealing and has been successful in prior studies. Quoting the performance of one system as a factor of another is common practice. It is common to see statements such as "System X is 10% faster than System Y," which implicitly assumes a product model.

In the multiplicative model, the definitions of the factors and residuals have an intuitive interpretation which an ACEC reader should understand so that the implications of the MEDIAN report are grasped. These are pointed out below:

- System_Factor :

- * The system_factor represents the inverse of the "average" speed of the system. The system_factor for system X is the ratio of the average speed for system BASELINE over the average speed of system X. (Where BASELINE is the compilation system arbitrarily selected to have a system_factor of one.)

The system factor will be less than one for systems faster than the BASELINE compilation system. Systems slower than the BASELINE will have values for system_factor larger than one. The system_factor reflects both software and hardware contributions to compilation system performance.

Small numeric differences in computed system factors should not be considered very important. The difference between factors of 1.05 and 1.00 may not even be statistically significant, although to determine this would require statistical inference which the median Polish model does not support. When comparing different implementations, it is important to remember that when a project selects a compilation system it will typically be used for several years spanning multiple releases of the compiler and the operating system. The relative performance of two different systems can easily change by more than 10% after each makes their next release. Two systems should be considered to be of "essentially equal performance" when the difference in system factors is less than X per cent (when X is a project defined tolerance which probably should never be less than 5%). Furthermore, users of the ACEC should remember that the system factors are based on a set of 1383 individual test problems covering all Ada language features, and may not emphasize the particular combination of features which will be stressed by any specific project.

– Problem_Factor :

- * The problem factor represents the "average" execution time for a problem. The scaling in MEDIAN results in a value for the problem_factor for test problem Y which represents the time it would take to execute that test problem on system BASELINE if the BASELINE system had its "average" performance on problem Y. The problem factor is calculated to reflect the "average" performance of all systems on test problem Y, and when the BASELINE system optimized problem Y more than the other systems, the problem_factor will be larger than the measured times.

– Residual :

- * The residual represents the difference between the fitted model and the raw data.

A small residual value (less than 1.0) for system X on problem Y indicates a test problem where system X executed much better than the other systems. Where system X generates better optimized code than the other systems for problem

Y, a small residual will result. A large residual value (greater than 1.0) indicates a test problem where system X executed relatively worse on problem Y — it can indicate a test problem where all the other systems generated optimized code for the problem Y, but where system X did not.

Test problems which translated into nulls but where noise in the measurement process resulted in a small nonzero time can produce residuals which are very different from one without being of any real significance.

A test problem where all residual values are close to one does not imply that none of the systems optimized the test problem. This simply indicates that there was little difference between the systems. All the compilation systems might have produced optimal code for the test problem Y, but this information cannot be derived from examination of the residuals for test problem Y. If a reader wants to know if all systems optimized problem Y, they must examine the related tests, or examine the "raw" data.

Keeping in mind the implications of these definitions will help an ACEC reader in reviewing this guide, and in understanding the results of a MEDIAN report.

6.2 STATISTICAL MODEL

This is the theoretical justification for the statistical model that ACEC uses.

The statistical model assumes that measured test problem times are independent. Based on the construction of the test suite, this is not completely true, although it should be a good approximation. Several factors lead to non-independence of test results, including:

- There are sets of related test problems whose performance will be correlated. For example, a system with slow subprogram calling conventions will be slow on all test problems that call subprograms. The timing will not be statistically independent.
- Consider a single compiler run on two different hardware processors of the same family — such as two different 1750A processors, or a VAX 11/780 and a VAX 8600. The performance difference will be highly correlated, one being approximately a factor of the other reflecting the overall higher speed, with small differences reflecting differing ratios of processor to memory speeds or the presence of relatively faster (slower) floating point processing units. The sets of timing data will not be statistically independent.
- Consider the performance changes likely to occur between successive releases of a compiler. Many test problems will generate the same code in each release, introducing statistical dependencies.

- The problems which test the **DELAY** constructions are somewhat special. It is not true that a good system should execute these statements as quickly as possible — even the most optimizing system should not execute these statements any faster than the value specified in the **DELAY**. The performance measurements for the **DELAY** problems will not be used in the statistical analysis.

A least squares approach to fitting a product model (Two-way Analysis of Variance, ANOVA) is numerically sensitive to the presence of statistical outliers. The fit can be greatly changed by the presence of a few data points “way off” the norm. Given the large number of test problems and the very different approaches to the implementation of several of the Ada language constructions which have been observed, a robust statistical procedure was adapted for MEDIAN.

Another approach is to normalize the timing data by selecting (arbitrarily) one system as a base and dividing the times of all other systems on that problem by the time of the baseline system for that problem. That is, the normalized time will be:

$$N (\text{System} , \text{Problem}) = \frac{T (\text{System} , \text{Problem})}{T (\text{System-Baseline} , \text{Problem})}$$

where $N (\text{System} , \text{Problem})$ is the normalized figures of merit, $T (\text{System} , \text{Problem})$ is the observed time to execute the “Problem” on the “System”, and “System Baseline” is an arbitrarily selected system. This approach is presented in the following references:

- “Re-evaluation of RISC I” by J. L. Heath in Computer Architecture News, Volume 12, Number 1, March 1984
- “A Performance Evaluation of The Intel iAPX 432” by Hansen, Linton, Mayo, Murphy and Patterson in Computer Architecture News, Volume 10, Number 4, June 82
- “How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results” by P. Fleming and J. Wallace in CACM, Volume 29, Number 3.

Consider a simple numerical example with two systems (S1 and S2) and two problems (P1 and P2) with measurements as below:

	S 1	S 2
P1	1.0	2.0
P2	2.0	1.0

Normalization on a system will produce either

	S 1	S 2
P1	1.0	2.0
P2	1.0	0.5

or

	S 1	S 2
P1	0.5	1.0
P2	2.0	1.0

depending on which system is selected as a base. The resulting matrix is unfortunately asymmetric. Whichever system is chosen as a base, the other system will have a performance factor of 1.25 — the mean is $1/2 * (0.5 + 2.0)$. That is, the method chooses a different system as "fastest" depending on the system selected as a base. Using harmonic means to compute factors, as discussed in the Fleming and Wallace paper, would help some, but would still produce an asymmetric table.

Where the baseline system does a particular test problem exceptionally well, this will only be apparent by observing that the ratios for the non-baseline systems are consistently lower than the ratios for other test problems. If there are some test problems for which the baseline system does not have measurements this normalization will break down. Test problems optimized into nulls must be treated specially to avoid dividing by zero.

6.2.1 Median Polish

MEDIAN performs a multiple cycle median Polish fit to a product model, as described in Applications, Basics and Computation of Exploratory Data Analysis, by Velleman and Hoaglin, Durbery Press, 1981. This iterative procedure is a statistically robust process in that the fit is not distorted by a few extraordinary values. Consider a two-dimensional array of measurements with a row for each problem and a column for each system. The analysis proceeds by taking the logarithms of the measurements to make the additive fit of the median Polish process reflect the product model. Then, if the analysis is to be done row-first, the median of each row is computed and subtracted from each element in the row, and added to the row factor for that row. The corresponding computation is done for each column. If the analysis is to be done column-first, then the medians are first computed for each column, and then for each row. These computations constitute one cycle in the median Polish fit. The cumulative corrections are the factors. The values in the array are the residuals. Large residual values for a test problem show where a system was slower than predicted, given the performance of that system on all other problems and the performance of all other systems on that problem. Similarly, small residuals indicate faster than expected performance. Missing data points are ignored in the calculation of medians for the row and columns.

A median Polish analysis can produce different results when the row/column order is varied — a row-first analysis does not always produce the same results as a column-first analysis. The ACEC uses a column-first analysis, because there are considerably more test problems than there are compilation systems. The median of a small set (less than 5 systems) is more sensitive than the median of a large set (over a thousand problems) — the difference between the median and the next larger (or smaller) element in a set of five can be (and has been observed on the ACEC trial systems results) a much larger difference than the difference between the median over the problems and the next larger (or smaller) problem. It would be possible to perform a median Polish analysis both row-first and column-first and permit users to examine both sets of results, and an interested ACEC user could perform this adaptation by interchanging the order of the two procedure calls in the procedure `CALCULATE_MEDIAN_POLISH_DATA_FIT`.

The MEDIAN tool does not perform alternative analysis and present results from each. Studies were performed during development using a row-first median Polish, a column-first median Polish, and an ANOVA technique modified to deal with missing data. In many cases the different analysis produced similar results. They differed significantly when dealing with a set of data which had four systems which were fairly similar in their approaches (as revealed by the residuals and small differences in the different analysis approaches when the four were analyzed separately) and one system which was very

different. The “different” system had a highly non-Gaussian histogram, and the different system factors produced for this system are understandable because it is difficult to select a single system factor to match its residual patterns — different analyses essentially gave different weights to fitting the relatively fast or slow test problems on this system. None of the analysis approaches can be considered “wrong” and when they differ greatly it indicates that the underlying data being analyzed is abnormal.

Because this case was somewhat anomalous, and because of a desire to minimize the complexity of interpretation of analysis results, the MEDIAN only performs one analysis.

For the example data discussed above, MEDIAN would produce equal system and problem factors, and the residual matrix would be symmetrical. Both system factors will be 1.0, and both problem factors will be 1.41 — 1.41 is an approximation to $\sqrt{2.0}$, and 0.71 is an approximation to $1.0 / \sqrt{2.0}$.

	S 1	S 2
P1	0.71	1.41
P2	1.41	0.71

Test problems which have either been optimized into nulls and take zero observable time (or space) must be given special treatment since the logarithm is singular at zero. MEDIAN ignores these observations. Systems will have very similar performance, with a test problem which every system failed (or every system translated into a zero time noop), not producing a problem factor.

One system — the first one listed in the enumeration type defining the names of the systems in the package MED_DATA — is arbitrarily chosen as a base. The other factors are scaled accordingly. This choice of base produces problem factors which would be the observed measurements if the tests were run on a mythical “average” compilation system with the same system factor as the baseline system, but with all residuals having a value of 1.0. Programmers developing portable software can use these factors as an “implementation independent” measure of the costs of various constructions.

6.2.2 Outliers

MEDIAN flags outliers in the residuals by using the rule of thumb suggested in Applications, Basics and Computation of Exploratory Data Analysis. That is, values greater than the

upper quartile plus three times the spread between the quartiles, or less than the lower quartile minus three times the spread, are considered to be statistical outliers. For comparison purposes, MEDIAN computes the residual mean and standard deviation and the values corresponding to the mean plus or minus both one and two standard deviations. MEDIAN also prints for each system and each problem, the extreme values of the residuals, the interquartiles, and the medians. The "spread" between the best and the worst residuals for a problem is a measure of how far apart the different system approaches to the problem are. When performance measurements are available for a test problem on a "large" number of systems, a large spread suggests that the slower systems could be improved. The spread of interquartiles for a problem is a good measure of the "variability of approach" of a problem since it is not determined by one system being exceptionally good (or bad).

Any test problem which has a large residual factor and which is flagged as an outlier, should be examined to see what language and hardware features are being used. It is helpful to understand what features of the test problem made the execution time appear unusual. For example, the 1750A processor has a special instruction to test if a register value lies between two limits (CBL), which is clearly useful for testing range constraints. Other target machines may have to test a range with a pair of compare instructions. This is a reflection of the fact that the ACEC is an estimator of a total target system performance, both hardware and software. A good optimizing compiler for a target machine which does not have a similar capability will compare poorly to compilation systems on targets which have (and use) the capability.

6.2.3 Optimization Analysis

Consider the test problem SS41 [$ii := 1+1;$], which tests folding of integer expressions. If most of the compilers tested perform folding, the residuals of systems which fold will be close to one, and compilers which do not fold will show up as slow (large residuals). However, if most compilers tested do not fold, the residuals of non-folding compilers will be close to one and compilers which fold will show up as fast (small residuals). Simply by examining the residuals, it will not be possible to tell if most compilers perform the optimization or not. To determine this, it may be necessary to examine the code produced, although it will usually be possible to tell by examining the measured times against other statements. (The SSA report looks at optimizations by examining related tests.) For the cited example, the time for SS7 [$kk := 1;$], will be equal to the time for SS41 if folding occurred. Similarly:

- The time for SS56 [ii := ll; ii := mm;] will be equal to that of ss11 [kk := ll;] when dead assignments are optimized away.
- The time for SS8 [kk := int(1.0);] will be equal to the time for SS7[kk := 1;] when compile time conversion is performed.
- The time for SS142[xx := max2(yy,zz);] where max2 is an inline maximum function, will be the same as the time for

SS144[if yy >= zz then xx := yy; else xx := zz; end if;]

when inline processing is done well.

- The time for SS168[e1(ei) := one;] with full range checking, can be compared to the time for SS17 (the same statement with checking suppressed), to observe the cost of range checking on one dimensional arrays.
- The time for SS54[ii := i1(ei+1);] will be the same as the time for ss53[ii := i1(ei);] if the compiler folds a constant term in an addressing computation into the base address of the array, as is possible on non-descriptor based targets.
- The time for SS93[if false then die; end if;] can be compared to a null statement to see if the compiler does control flow optimization and removes unnecessary code.
- The time for SS210[xx := (yy*zz-0.125) / (yy*zz);] will be the same as ss211[xx := yy*zz; xx := (xx-0.125)/xx;] when common subexpressions are processed. If the time for SS643[xx := (yy*zz-0.125) / (zz*yy);] is also the same, then the compiler is performing canonical ordering to recognize common subexpressions. It requires additional compile-time processing to recognize that yy*zz is equivalent expression to zz*yy.
- The times for SS413[xx := sgn(yy)+1.7;] and SS414[xx := 1.7+sgn(yy);] will be the same if order of expression evaluation is performed; otherwise, ss414 may be slower due to saving and restoring the value 1.7.
- The times for SS423 and SS424.

```
[ss423]      isum := 0;
              for i in 1..ten
                loop
                  ii:=i*2;
                  isum:= isum+ii;
                end loop;
```

```
[ss424]      isum:=0;
              ii:=0;
```

```

for i in 1..ten
loop
  ii:=ii+2;
  isum:=isum+ii;
end loop;

```

SS424 is a strength reduced version of SS423 where a multiply was reduced into an add. The execution times of the two problems will be close if strength reduction is performed.

- The time for SS212[for i in 1..10 loop xx := yy; end loop;] will be the same as the time for SS3[(xx := yy;) if loop invariant motion and omission of irrelevant code is performed.
- If the time for SS45[ji := 0;] is much faster than the time for SS11[kk := ll;] or SS7[kk := 1;] then the system is probably using a special machine idiom to set a memory location to zero. If the time for ss207[if ll < 0 then die; end if;] is significantly faster than the time for SS208[if ll > 100 then die; end if;] then the target processor may have a load instruction which sets the condition codes to reflect a comparison against zero, permitting the compiler to generate a sequence of instructions (LOAD/CONDITIONAL_BRANCH) rather than a sequence (LOAD/COMPARE/ CONDITIONAL_BRANCH). Even on targets where every load does not set the condition codes, a comparison against zero may be faster than a general comparison.

This list is not exhaustive. Related tests are listed in the results file and in the comment template in the source file.

6.2.4 Analysis of Results

For large test problems, it is hard to tie performance problems to a specific language construction. If the Whetstone benchmark program is relatively slow, it is not easy to know why, without a detailed examination of the generated code. It could be due to procedure calling, parameter passing, array subscripting, math library routines, arithmetic computations, loop constructions, or any combination of the above. The test suite contains many small tests of particular features so that specific problem areas can be identified. Larger test problems combine the features in ways that reflect typical usage, and provide samples of code containing sequences of language feature usage. This permits the exposure of the interaction between features, and of the dependence of the performance of a construction on its context.

Not too much emphasis should be placed on any one test problem. Even ignoring the possibility of measurement errors, large residuals may be due to "unusual" interactions with the system rather than inherent properties of the system and the problem. For example, a test problem to measure the speed of access to intermediate scoped variables might be unusual due to the presence of extra instructions to set up a base register to maintain addressability in an S/360-like architecture. An unusual timing measurement should be viewed as an indication of the need for further study. It may be necessary to examine the machine code produced to see if a problem is due to peculiar system interactions not directly tied to the features used in the particular test problem.

The system factors give a measure of overall performance with respect to the workload defined by the test problems in the test suite. Readers who are interested in a different workload, should pay more attention to the test problems which reflect their interests. For example, where a reader is not interested in either tasking or fixed point operations, the results of test problems for these features can be ignored. If the total number of tests not of interest to a reader is large, it might be best to rerun MEDIAN ignoring these tests.

All readers can get value from an examination of the test results to determine if, on the systems of their choice, there are any language constructions which they would want to avoid for performance reasons. If a system has a particularly bad performance on a feature of critical interest, it might be necessary to base the choice of a system not on fastest average performance, but on one which insures that the worst case performance will be tolerable.

MEDIAN prints the following output:

- The matrix of measurements with rows and columns labeling the problems and systems they represent,
- A histogram of the (log) residual values (for a visual comparison to a Gaussian distribution),
- The matrix of residual factors (including the computed system and problem factors),
- Statistical summaries of the set of residuals. These include a Kolmogrov-Smirnov statistic comparing the distribution of residuals with a Gaussian distribution, and a Coefficient of Determination test (analogous to linear regression models) to show how well the product model explains the variations in the observed measurement data.
- A summary of the residual data (extremes, interquartiles, medians),
- A list showing the problem name, minimum, maximum, interquartiles, median, and

interquartiles spread, for each problem where either the upper or lower interquartile is large enough to be a statistical outlier, and

- A list of the summary data for each system.

The report produced by MEDIAN is tabular. If more systems are compared than fit across one line of output, the program will output additional pages as needed. MEDIAN does not write more than 80 columns on a line unless this value is modified by a user, as described in the User's Guide, Section "RUNNING MEDIAN".

6.2.5 Summary

The proposed data fitting technique has important properties. To summarize:

1. The residual matrix identifies strong and weak points in compilation systems. For test problems which consist of examples of code fragments constructed to be optimizable by particular techniques, the residuals will show where some systems optimize the code and some do not.

MEDIAN permits a form of report reading by exception. A reader can focus attention on test problems with large (or small) residuals, since these problems characterize where the performance of one system differs sharply from the other systems tested. With many test problems in a test suite, this automated assistance in identifying "interesting" problem results is important.

2. The procedure is statistically robust. The fit is not greatly modified by large changes in a few test problems.
3. Sample data can be distributed without revealing the details of any particular implementation. The problem factors and the interquartile spreads for each problem as computed over trial systems will not reveal detailed timing information on any particular implementation.

MEDIAN assists the ACEC user in satisfying the following high-level requirements.

1. Compare the performance of several implementations.
2. Isolate the strong and weak points of a specific system.
3. Determine what significant changes were made between releases of a compilation system.
4. Predict the performance of design approaches.

The raw data matrix of measurements can be directly inspected to answer any specific performance questions. Summary statistics on overall performance characteristics are computed by the analysis program MEDIAN. These statistics assist the end user in interpreting the significance of the measurement data.

Where measurement data from several systems are available, MEDIAN computes system factors reflecting overall system performance and the residual matrix which gives insight into the performance characteristics of a system. The residual matrix will flag test problems where a particular system performed significantly better or worse than expected, relative to the average of that system over all problems, and to the average of that problem over all systems. This flagging isolates the test problems where a system performs either strongly or weakly. Users can then examine the constructions and features used in these problems to see why the system behaves anomalously on the problem.

MEDIAN is a program to compare data between systems. To use MEDIAN to determine aspects of performance of a system for an end user who is testing only one system, a set of sample data is distributed as part of the ACEC product. This sample data set is the problem factors computed by analyzing the results of the five systems on which the ACEC test suite was executed during development to verify the portability of the ACEC and the support tools. This sample data set will be updated with each release of the ACEC, using the then current releases of the Ada compilation systems. It will track the average Ada performance of systems over time. *Because of the scaling effect of selecting the first system named in the enumeration type as unity*, the sample data is essentially an estimate of what performance would be from a compilation system which had "average" performance on all test problems when executed on a processor with the same speed as the baseline system. As an example, a DEC VAX Station 3100, which is approximately a 2.7 million instruction per second (MIP) machine, was used. Executing MEDIAN with these two sets of measurement data — the sample set and that collected from the one system the user tested — permits isolation of weak and strong points relative to the "average" implementation as observed in the trial systems.

Presenting the measurements from successive releases of an Ada compilation system to MEDIAN highlights the differences between the two releases. Test problems using language features which were changed between releases should produce different measurements. Where these differences are large, MEDIAN will flag the test problems. Where a large difference is observed in a test problem which uses many language features, it may not be obvious why the difference occurred. The ACEC contains many small tests which should isolate differences, and a large test problem can be studied to see what features it used. Hopefully, that will be sufficient to explain the performance of large test problems.

Information permitting estimation of the performance of different design approaches can be obtained by studying the actual results. The tests permit the observation of performance of some language features, such as rendezvous, which might be too slow to permit their frequent inclusion in some applications. The coding style tests are intended to be compared against other tests which perform the same function in a different manner. Comparing the results can suggest the faster alternative for a particular target system. The ACEC provides information to permit users to make such decisions in an informed manner.

6.3 TIMING DATA

This section discusses how well the product model assumed by MEDIAN can be expected, in principle, to fit measurements of execution time.

The product model will be satisfied when there is a time associated with each language construction, and the total time for a problem is the sum of the times for the constructions it uses. For non-optimizing compilers, this assumption will be satisfied.

For optimizing compilers, the linear assumption might break down. Many optimizing transformations — common subexpression elimination, folding, load/store suppression, most machine idioms, strength reduction, and automatic inline expansion of subprograms — will not modify the asymptotic complexity of the generated code. For a linear sequence of Ada statements, the code generated by either an optimizing or a non-optimizing compiler will be of a linear complexity — the optimizing compiler will have a smaller coefficient. This is not true for loop invariant motion where the difference between an optimizing compiler which moves an expression evaluation out of a loop and a non-optimizing compiler which does not is determined by the number of loop iterations, rather than being a property of the optimizing techniques. For most optimizing techniques, the difference in execution time between optimized and non-optimized translations will be some factor based on the code generation approach and whether the test problem contains constructions which are amenable to optimization.

Some test problems are not amenable to optimizations. Consider two compilation systems for the same target machine, one which does straightforward code generation and one which produces optimal code. There are some test problems where the straightforward code will be the best possible code for the target machine. The optimizing compiler will not be able to produce any better code for that problem. The analysis program will show the optimizing compiler as executing slower than expected for these simple test problems. In this example, the real problem the optimizing compiler has is that it cannot do any

better than the best code for the target machine, but based on the average performance on the other test problems, it is expected to.

A product model has been widely applied and successful in similar studies. Citing the performance of one compilation system as a factor (or percentage) of another is a common practice and implicitly assumes a product model. Replacing the target hardware with a processor which is uniformly some multiple of the original should intuitively result in a proportional change in the evaluation of the compilation system on the new processor relative to the old one — this also assumes a product model.

A product model is simple and intuitively understandable. A detailed examination of the model assumptions and sample results suggest that a perfect fit should not be expected. This lack of perfection is not a fatal flaw. A main purpose of the statistical analysis is to determine and present underlying trends in the large volume of performance data. In addition to fitting the data, the MEDIAN program will perform some statistical tests to see how well the data conformed to the model's assumption that the (ln) residual will be normally distributed. This is tested with a Kolmogorov-Smirnov comparison against a Gaussian distribution and a coefficient of determination computation to show how much of the variation in the data the statistical model explains.

Tasking tests executed on multiprocessor target systems which assign separate Ada tasks to separate hardware processors are measuring a conceptually different quantity than the same programs when executed on a single processor system. The use of priorities to control whether the task performing the ACCEPT or the entry call arrives at the rendezvous first may not work on multiprocessor targets.

6.3.1 Extrapolation of Timing Data

An ACEC user might want to extrapolate measurements from one member of a family of processors to other members in the family. For example, data may be available for Vendor One's system executing on a VAX-station 2000 and Vendor Two's system executing on a VAX-8800. MEDIAN will numerically compare the two sets of measurements, but this result confounds the machine effects with the compilation system effects. The question the user would like to have answered is: "If both vendor systems were run on the same hardware, which would be faster?" The performance difference between hardware speed of two members of a family is rarely *exactly* linear. The relative speeds of floating point processing, cache memory systems, instruction prefetching, software simulation of some instructions, etc. usually result in some instruction sequences performing faster or slower than the average difference between the two processors. If a user is willing to accept that one member of a family is X times faster than another member of the family

(or equivalently, that each has a given MIP rate which is assumed to be applicable for the applications of interest), then a user could compare the system factors computed by MEDIAN with the relative hardware speeds to see how well the two performance estimates agree. Large differences between systems will show up this way, however users need to be aware that hardware options (such as the different floating point hardware support as provided for several families of machines) can complicate such direct comparisons. If a user needs to know performance on a particular target configuration, the ACEC should be run on that configuration. Extrapolating from "similar" systems can introduce errors.

Dividing each test measurement by the target hardware's average instruction time will estimate the (fractional) number of instructions to execute each test problem. Some people have been tempted to use this measure to "factor out" target machine hardware characteristics and measure the compiler code generator (independent of the target hardware). This is not really useful for two reasons.

- It is difficult to determine target hardware scaling factors.

Different Instruction Set Architectures take different numbers of instruction to perform similar tasks. Assessment of "overall" processor speeds requires a determination of the appropriate mix of instructions and memory characteristics (cache faults, wait states, prefetching, etc.).

Simply counting instructions can be misleading because a RISC ISA will typically require more (but simpler) instructions to translate a test problem than a CICS processor. If a RISC processor can execute its simple instructions quickly and cheaply, it may be a more cost effective design.

There have been many compilers released which do not use all the features of the target machine. An MC-68020 is upward compatible with an MC-68000, so a compiler which treats an MC-68020 as an MC-68000 will work. If a "hardware factor" is determined assuming that all features are used, it will give misleading results.

- It is not helpful to do so.

The point of the ACEC is to evaluate compilation systems, including both software (compilers and runtime libraries) and hardware — not compiler writers. Target architectures make a significant contribution to overall system performance which should not be ignored.

In an academic environment, it might be desirable to give higher marks to a highly optimizing compiler generating code for a feeble target processor than to a straightforward compiler producing fair code for a capable target processor. However, the purpose of the ACEC is to provide a capability to evaluate Ada implementations for their suitability for MCCR applications. If the first complete system (optimized

compiler / slow hardware) is slower than the second (straightforward compiler / fast hardware), that is the determination the ACEC is interested in making. A project may well select a slower system if its performance is adequate, but it would be foolish for the ACEC to report that the first system is *faster*.

6.3.2 Tuning for Implementation Dependencies

Compilation systems may have implementation dependent facilities which can be used to improve the performance of the generated code. Their use is not portable, and so results are not necessarily comparable between systems.

The syntax and semantics of the string which is passed as an actual FORM parameter for file OPEN and CREATE operations is defined by each implementation. On some systems, there may be settings which produce much faster performance than the defaults. For example, there may be options for specifying shared or exclusive file access, contiguous disk allocations, the size of the initial and secondary extents, multiple buffering of sequential files, suppression or enabling of read-after-write checking for physical I/O operations, the physical properties of the disk device (seek times, rotational latency, error recovery procedures), ... These can all have major performance impacts. The ACEC is designed for portable operations, and uses the default settings. To get an idea of the difference tuning can make in performance, it is possible for ACEC users to experiment with the various options provided and select the ones which provide the best execution time and satisfy the program's requirements for I/O functionally. When this has been done, readers should be aware of the modification and interpret the results accordingly — in particular, remember that not all systems may have had the same degree of performance tuning.

Once a project has selected a compilation system for use, they may be very interested in the performance changes resulting from modifying FORM parameters. ACEC users should feel free to explore such modifications, but they should consider the modified test problems to be "new problems" which do not replace the original, portable version distributed. Results on modified problems can be very interesting, and important for best exploiting a system.

In addition to FORM parameters, there are other special features which an implementation may provide which can enhance performance. Special pragmas are the feature which is the most likely to occur. Again, an ACEC user may modify some problems to observe the effect of non-predefined pragmas, but they should consider these to be "new problems" which are not necessarily comparable between different systems.

6.4 CODE EXPANSION

This section discusses how well the product model assumed by MEDIAN can be expected, in principle, to fit measurements of code expansion size.

To compare code expansion sizes for different systems, the user will run `FORMAT` on the performance test log file for each system. `FORMAT` will produce two output files for each system, one containing code expansion sizes for each problem run on the system, and the other containing execution times. The user will run `MED DATA CONSTRUCTOR`; the input files will be the files containing the code expansion sizes for each system. `MED_DATA_CONSTRUCTOR` will produce a version of the `MED_DATA` package containing code expansion sizes, which will be compiled to provide input to `MEDIAN`.

The product model assumption will be satisfied for non-optimizing compilers which have a size associated with each language construction. The problem factors computed by the application of the median Polish algorithms will be the estimates for these sizes. For optimizing compilers, as with timing measurements, it is not reasonable to assume that the total code expansion size will be a simple sum of the features used; however, as with performance optimization, code space optimization should be linear with respect to the constructions used. Removing unreachable code is the primary space specific optimization, which will be linear. Other optimization techniques should match the *product model nicely* — loop invariant code motion does not modify space measurements as it does timing measurements.

The code expansion measurements in the ACEC reflect the difference between memory addresses at the beginning and end of the code fragment bracketed by the timing loop. This could introduce errors in the measurements if the compiler generated machine code for the Ada text within the timing loop in different address ranges. For example, some compilers may allocate code generated for generic package instantiations to different control sections. Compilers which produce the same amount of code could have very different measurements depending on where the space is allocated. The ACEC tries to avoid this problem by minimizing the use of declarative regions within the timing loop.

Where the code bracketed by the timing loop code is a call on a subprogram, the space measurement will not reflect the space associated with the referenced subprogram. The execution time for the subprogram will be accounted for in the timing measurements. This discrepancy between time and space measurement should not confuse a reader.

Consider a system which translates a language construction by a call on an RTS routine where the RTS routine is not loaded unless the language feature is used in the program. If total program size, including all the loaded RTS routines, were considered, it would not be reasonable to expect a linear factor to adequately model usage. A call on the

support routine will take "n" words of code, and the RTS routine takes "m" words when loaded. If the feature is referenced once in a program, it will take "n+m" words; if the feature is referenced twice it will take "2*n+m" words; if referenced "p" times, it will take "p*n+m" words. While this is asymptotically linear in "p", for small programs (small "p"), the non-linear behavior introduced by the "m" will thwart attempts to fit one linear factor to the language construction. This problem is avoided by measuring code expansion sizes rather than total program size.

Because of the measurement technique used to measure code expansion size, subtracting addresses at the beginning and end of the timing loop, a few anomalous measurements might occur. If a compiler allocates the code within the timing loop to different control sections, it is possible that a test problem may have a code expansion measurement which is much smaller than the space actually allocated it (since the measurement misses space not contiguously allocated). It is conceivable that a negative space measurement may be reported if the end of the loop is allocated in memory with a lower address than the beginning of the loop. Outliers should be examined carefully.

A small code expansion size sometimes results when the generated code calls on runtime support routines rather than generating code inline. Expansion sizes and execution times may not be strongly correlated — sometimes small sizes will correspond to long execution times, because of loops in the test problem, calls on external routines (either in the runtime library or user defined subprograms), or slow instructions. Some correlations should be observed. A test problem which takes zero space should also take zero time.

6.5 RTS SIZE

This section discusses how well the product model assumed by MEDIAN can be expected, in principle, to fit measurements of the amount of memory space generated inline for each test problem.

The simplest implementation approach will result in an RTS package with a fixed size for all programs. The next higher level of sophistication will partition the RTS and only include in a program load image those pieces of the RTS which are needed to support features actually used by the program. Neither of these design approaches will result in data which can be accurately fitted by a product model. The RTS size will either be constant, or the sum of a sequence of terms which are either included or not included. A language construction can require the loading of an RTS routine, but a test problem which uses the construction twice should not load the routine into memory twice, which would be required if a product model were to match the data.

This expected behavior is not consistent with a product model. An additive model might better fit the RTS size data. The MEDIAN program can be adapted to explore an additive fit by avoiding the taking of logarithms and the scaling of factors.

MEDIAN should not be applied to runtime system size data. If available, RTS size data should be reported in tabular form or summarized as a range of values. It should be accompanied with a discussion of the process used to collect the data and of the efforts attempted to verify cross system comparisons.

On an embedded target, the size can determine if a given program with the accompanying runtime will load and run in the available memory. Whether a (potentially sharable) constant area in the RTS is counted is not important.

6.6 COMPILATION TIME

This section discusses how well the product model assumed by MEDIAN can be expected, in principle, to fit measurements of the time to compile and link each test program.

The time to compile and link each compilation unit is collected and analyzed. There is no extraction tool similar to FORMAT distributed with the ACEC to automate the collection of compile time data because timing facilities available in a command file to measure elapsed time and display results are dependent on the host operating system. If the example provided by the DEC Ada command files (CMP.COM, CMP_CK.COM, CMP SP.COM, and CMP_DIFF_NAMES.COM ...) can be readily adapted to a new host system, it will be a straightforward task to extract the compilation time information from log files. A programmable editor should be sufficient for this task, or a program could be written to extract the times. On other operating systems, it may be much more awkward.

The programs 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 times of these programs can give insight to the overall compilation rates.

The compilation times collected measure the time from source submittal to the construction of an executable load module. Link time can be a significant fraction of total time to transform the source of a compilation unit into an executable load module. The ACEC includes link time in the compilation rate measurements. This has not been traditional in the quoting of compiler performance, since many compilation systems for prior languages have used system standard linkers which are common to all language processors on a target system and are often written and maintained by different organizations from the one

writing the compiler. The Ada language definition requires checking of subprogram type signatures between separate compilation units, and while this could be done at execution time, it is possible (and more efficient) to do the checking at link time. Different Ada systems have partitioned the checking work differently between the compiler, the linker, and the execution time environment.

On a system with a linking loader which links modules only after a program execution has been requested, it will not be possible to separately measure link time, or to combine link time with compile times. The time associated with linking will appear to users as a slow program load. Linking loaders complicate measurement, however they are not a widely used implementation approach for Ada compilation systems. When testing such a system, an ACEC user will report on the observable times (compilation times for the test programs and execution times for the test problems) and ignore program load times.

The ACEC measures the time to compile and link various source files. This approach avoids non-productive controversy about how to count lines. Should comments and blank lines be counted? Should only executable statements be counted, and if so, should initialization in a declarative region be counted as executable statements?

There are several Ada features whose use may impact compile rates:

- Compilation unit size. Several code generator paradigms build a representation of a program (or a piece of a program such as a subprogram, basic block, linear sequence of statements, ...) and perform various manipulations on the structure to try to produce good code. Some of the algorithms used are of more than linear time complexity — they run much slower on large units than on short ones. Hopefully, in exchange for taking more time for compiling, better code will be generated.
The time associated with loading each phase of a compiler may mean that all programs take some fixed time. A short program may not take much less time than a slightly longer one.
- Presence of generic instantiation. The amount of time associated with instantiating a generic unit can be substantial. The compiler must check that type signatures match, and if it is treating instantiation as a form of “macro expansion” it may try to optimize generated code. When an actual generic parameter is a literal, the compiler may use this value to fold expressions in the generic body. Some Ada compilers treat a generic instantiation as a macro expansion and effectively recompile the source code when instantiating the generic unit, substituting the actual generic parameters — with the expected impact on performance.
- Each library unit referenced in a **WITH** clause can require considerable processing time. The time to search a program library to find the matching definition can be substantial, and can vary with the usage history of the program library.

- The placement of files for source, object, and temporary files on disk can produce better performance by reducing the time associated with physical I/O. Contiguous allocation of files on separate devices on separate channels would lead to best times. Contending users, or separate files on the same disk causing disk head contention, will all tend to degrade performance.
- Compilers are sensitive to various system tuning parameters, such as the amount of main memory, working set sizes, and contention for CPU and secondary storage.

When the MEDIAN program indicates that one compilation unit compiles much faster (or slower) on one system than on another, the reader may want to examine the compilation unit to see what language features are being used.

7 VERSION DESCRIPTION DOCUMENT (VDD)

The detailed description of each test problem will be included in the VDD. That document will contain the following appendices:

Appendix	Name	Contents
I	Test Problem Descriptions	List of test problem names with a brief description of each. New or withdrawn tests are identified.
II	Test Problem to Source File Map	List of test problems and the source file they are contained in.
III	Tape Description	List of files on the delivery tape.
IV	Quarantined Test Problems	Cross reference of test problems observed to fail on some systems.
V	ACEC Keyword Index - 1	List of primary purposes (with LRM references) and their associated test problems, as well as secondary, and incidental purposes, and comparison tests.
VI	ACEC Keyword Index - 2	List of test problems with their primary purposes (which may be for comparison with other tests).
VII	System Dependent Test Problems	List of test problems which exercise system dependent features.
VIII	Optimization Techniques	List of optimization techniques and the benchmarks designed to test them.
IX	Withdrawn Test Problems	List of test problems which have been withdrawn.

All these appendices will be useful to readers, especially in analyzing the MEDIAN output.

- Test problems marked as failed can be compared to the Quarantine list to see if they have failed on other systems, and to the system dependencies list. If these do not indicate prior failure, the description of the problem can be reviewed.
- Using the MEDIAN output, the test results flagged as outliers would be examined to determine any consistent trend. If a problem flagged as an outlier occurs in the

optimization techniques appendix, then the other problems listed in that appendix should be examined.

The ACEC is designed to be easily extendible. If an organization created their own unique test problems, or modified existing test problems, these new problems would not be described in ACEC's distributed documentation.

8 SIGNIFICANCE OF ERRONEOUS TESTS

Test problems which fail to execute will be flagged on the MEDIAN output. There are several reasons why problems may fail, not all of which are due to serious problems in the compilation system being tested. Reasons include:

1. The test problem could uncover a compiler error.
2. The test problem could produce incorrect code which fails at execution time.
3. The test could have been skipped and no attempt made to execute it. When new test problems are added to the test suite, this will be the status of those problems on the systems not yet tested. An organization with only a limited time for testing may not attempt to execute all the ACEC test programs. All test problems skipped for lack of time would be unavailable.
4. The test may be inappropriate. For example, the tests for file I/O are not appropriate for targets which do not support devices capable of maintaining a file system. The data for some system dependent test problems may not be available for the following reasons:
 - (a) The tests are not supported on the target. A test may use a floating point type with more precision than the target supports, or it may use an unsupported Chapter 13 feature, for example, several validated systems do not support the attribute label ADDRESS. There are a few test problems which require checking to be suppressed; SS242 refers to a discriminated record with an invalid discriminate and intentionally violates a range check, assuming it can reference an 8-bit field as if it were an unsigned byte with a range of 0..255.
 - (b) The tests require modifications which were not performed. An example is the tasking test problems which tie hardware interrupts to ACCEPT statements. These problems will require implementation dependent modifications to execute, assuming that the target system will support the feature at all. Until this modification is performed, they cannot be executed as intended. Similarly, the

test problem which calls on an assembler language program will need to be modified for each target to adapt to the interface conventions of the target.

5. A test may execute, but produce results which are considered unreliable. When the timing loop is unable to achieve the requested statistical confidence level within the maximum number of permitted outer timing loop cycles, the measurement is considered unreliable. The FORMAT tool which extracts measurements examines the confidence level indicator and, if it is set, creates a timing aggregate with a negative numeric value which is recognized by MEDIAN as indicating an unreliable measurement.

An unreliable measurement does not imply that the compilation system did not execute the test problem properly. It indicates that the timing loop did not measure the execution time reliably. Simply rerunning the test program is often sufficient to produce a reliable measurement — there may be less contention when the test is rerun.

9 UPWARD COMPATIBILITY OF THE TEST SUITE

Any test problem which retains the same name between releases of the ACEC shall be comparable. Modifications to a test problem between releases must not change the performance of that test problem. This convention will strictly limit the type and extent of possible modifications. The ACEC is adapted so that historical data from early releases of the ACEC will be comparable to the most current releases.

If and when a test problem is determined to be incorrect, and correcting it would imply a possible performance modification, the test will be withdrawn and a corrected version entered with a new name. The VDD Appendix IX, "WITHDRAWN TEST PROBLEMS" identifies all the withdrawn tests for a release.

10 ACEC USER FEEDBACK

The section describes the procedure to submit change requests and problem reports.

10.1 HOW TO REQUEST CHANGES

The procedure to request changes in either operations or in interpretation is the same. Readers and users may submit different types of requests: Readers would be likely to request modification to analysis output, or the addition of new test problems (or areas which should be tested). Users may request changes in the packaging of problems into programs, or modifications to control procedures.

The depth of detail of a change request may vary. Users may request the incorporation of a new test problem (which is submitted for consideration), or there may be a less specific request asking for more emphasis on some areas of concern. The more specific a request is, the easier it will be to respond to.

Change requests will be logged, evaluated and a determination will be made.

After completing the form on the next page, it should be mailed to:

R. Szymanski
Ada Compiler Evaluation Capability - Change Request
AFWAL/AAAF
Wright-Patterson AFB OH 45433-6543

ADA COMPILER EVALUATION CAPABILITY CHANGE REQUEST

ORIGINATOR IDENTIFICATION

Originator's Name -----
Organization -----
Address -----
Telephone -----
Date -----

SYSTEM IDENTIFICATION

ACEC VERSION -----
Compilation System Version -----
Host Operating System Version -----
Target Operating System Version -----
Hardware Identification -----
(if a test program is submitted for incorporation
into the ACEC, identify where it has been tested)

CHANGE DESCRIPTION AND JUSTIFICATION

(attach more pages if necessary)

10.2 HOW TO REPORT ERRORS

This section tells the user how and where to report problems with the ACEC. Problem reports will be logged, evaluated and a determination will be made.

After completing the form on the next page, it should be mailed to:

R. Szymanski
Ada Compiler Evaluation Capability Error - Report
AFWAL/AAAF
Wright-Patterson AFB OH 45433-6543

ADA COMPILER EVALUATION CAPABILITY SOFTWARE PROBLEM REPORT

ORIGINATOR IDENTIFICATION

Originator's Name -----
Organization -----
Address -----
Telephone -----
Date -----

SYSTEM IDENTIFICATION

ACEC VERSION -----
Compilation System Version -----
Host Operating System Version -----
Target Operating System Version -----
Hardware Identification -----

PROBLEM DESCRIPTION

Source File with Problem -----
Explanation: -----

(attach more pages if necessary)

11 NOTES

This section contains information only and is not contractually binding.

11.1 ABBREVIATIONS, ACRONYMS, AND THEIR MEANINGS

ACEC	Ada Compiler Evaluation Capability
ACM/SIGAda/ARTEWG	Association for Computing Machinery / Special Interest Group on Ada / Ada RunTime Environment Working Group
ACM/SIGAda/PIWG	Performance Issues Working Group
ACVC	Ada Compiler Validation Capability
BMA	Boeing Military Airplanes
CACM	Communication of the Association for Computing Machinery
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSCI	Computer Software Configuration Item
DEC	Digital Equipment Corporation
DMA	Direct Memory Access
DoD	Department of Defense
Hz	Hertz
I/O	Input / Output
ISA	Instruction Set Architecture
JSB	VAX instruction mnemonic
KS	Kolmogrov-Smirnov, statistical test
LRM	(Ada) Language Reference Manual (MIL-STD-1815A)

MCCR	Mission Critical Computer Resource
MIP	million instruction per second (MIP) (a measure of hardware speed)
MIS	Management Information System
RISC	Reduced Instruction Set Computer
RTS	RunTime System
SPR	Software Problem Report
t-test	Student's t-test, statistical test
TCB	Task Control Block
TOR	Technical Operating Report
URG	Ada Uniformity Rapporteur Group
VAX	Virtual Address eXtension (DEC instruction set architecture)
VDD	Version Description Document
VLIW	Very Long Instruction Word
VMS	Virtual Memory System (DEC VAX Operating System)
z-test	statistical test