Technical Report

CMU/SEI-89-TR-013
ESD-TR-89-021

Ada Adoption Handbook: Compiler Evaluation and Selection

Version 1.0

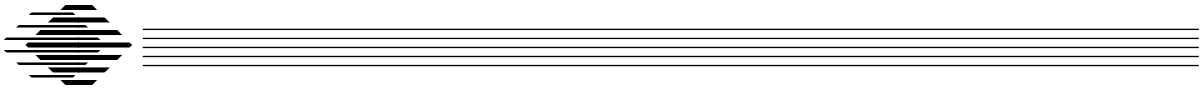
Nelson H. Weiderman

March 1989

Technical Report

CMU/SEI-89-TR-013
ESD-TR-89-021
March 1989

Ada Adoption Handbook: Compiler Evaluation and Selection
Version 1.0



Nelson H. Weiderman

Real-Time Embedded Systems Testbed Project

Unlimited distribution subject to the copyright.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

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

FOR THE COMMANDER

(signature on file)

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

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

Copyright © 1989 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

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

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX: (304) 284-9001 / World Wide Web: http://www.asset.com/sei.html / e-mail: webmaster@www.asset.com

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

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder. B

Ada Adoption Handbook:
Compiler Evaluation and Selection

Abstract: The evaluation and selection of an Ada compilation system for a project is a complex and costly process. Failure to thoroughly evaluate an Ada compilation system for a particular user application will increase project risk and may result in cost and schedule overruns. The purpose of this handbook is to convince the reader of the difficulty and importance of evaluating an Ada compilation system (even when there is no freedom of choice). The handbook describes the dimensions along which a compilation system should be evaluated, enumerates some of the criteria that should be considered along each dimension, and provides guidance with respect to a strategy for evaluation. The handbook does *not* provide a cookbook for evaluation and selection. Nor does it provide information on specific compilation systems or compare different compilation systems. Rather it serves as a reference document to inform users of the options available when evaluating and selecting an Ada compilation system.

1. Introduction

An Ada compilation system includes the software required to develop and execute Ada programs. Important components of the compilation system are the compiler, the program library system, the linker/loader, the runtime system, and the debugger. Evaluation and selection apply to the entire package, not just the compiler. In this report, the word "compiler" is sometimes used for "compilation system" for the sake of brevity, but the context should indicate the intended meaning. Another convention of this report is referring to the *Reference Manual for the Ada Programming Language* as *RM*, rather than *LRM* or *ARM*.

1.1. Purpose and Scope

> *Department of Defense directives require the use of Ada for mission-critical, embedded applications. This handbook presents information on strategies and techniques for selecting an Ada compilation system that is appropriate for a particular application.*

Department of Defense (DoD) policy on the use of Ada is specified in DoD directives 3405.1 [United States Department of Defense 87a] and 3405.2 [United States Department of Defense 87b]. The first directive describes computer programming policy in general and specifies that Ada "shall be the single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system." The second directive specifies the policy for weapon systems in particular. It requires the use of a validated Ada compiler, the use of software engineering principles facilitated by Ada, and the use of Ada as a program design language (PDL). Waivers are provided for in these directives, but will be increasingly difficult to obtain.

With regard to application areas not included in the paragraph above, DoD directive 3405.1 states: "Ada shall be used for all other applications, except when the use of another approved higher order language is more cost-effective over the application's life cycle, in keeping with the long-range goal of establishing Ada as the primary DoD higher order language (HOL)." By most standards, the requirements of embedded weapon systems are more demanding than those of management information systems (MIS). This handbook addresses both application areas, but with an emphasis on embedded systems. When MIS-type applications have special requirements, they will be noted.

There are risks in using any new programming language, particularly when new software engineering techniques are being adopted at the same time. But there are also risks and costs associated with failure to adopt modern technology. In order to reduce the risk of a new language, as well as the risk of immature compiler implementations, the selection process must identify key criteria and test the candidate compilation systems against the criteria. Even when there is only one compilation system available, or when the choice of compilation system is mandated, the risks of using a particular system should be identified so that the impact on costs and schedules can be better predicted.

The purpose of this handbook is first to convey to the reader the importance of evaluating an Ada compilation system with respect to application requirements and second to provide the necessary information and pointers to information to facilitate this evaluation and selection process. The handbook is not meant to be a cookbook. There are no simple answers because each application is different. There is no test suite or checklist that is sufficient for everyone or every project. The best that this handbook can do is identify the areas of importance to a reasonable degree of detail, provide the foundations to build an evaluation capability, and encourage the reader to follow some of the paths to more comprehensive information.

The handbook may leave the reader with the impression that the evaluation of an Ada compilation system is a daunting task. Since Ada was proposed with portability and uniformity in mind, it might be supposed that all compilation systems are the same. The fact is that the implementors were given freedom in many areas by the language designers, so that the language could be tailored to application needs. This has brought about variations in implementations, but for good and understandable reasons. These variations are, in part, responsible for the fact that Ada compilation systems are available for well under one thousand dollars and for well over one hundred thousand dollars. The costs of a good evaluation and selection process are considerable, but the costs of an inadequate evaluation are greater and can cause disastrous results for the system under development.

Information contained in the handbook is derived, in part, from the experience of the SEI Real-time Embedded Systems Testbed (REST) Project. The project was initiated in October, 1986, and as of early 1989 has had experience with three host systems, four target systems, nine Ada compiler vendors, and five Ada test suites.

The handbook is written for anyone who may be in the position of evaluating or selecting an Ada compiler for a project, or anyone who is managing a project for which an Ada compiler

must be selected. The handbook assumes that the reader has a general working knowledge of compilers, linkers, loaders, and Ada program library systems. The more general information for project managers appears in Chapters 2, 3, and 4. More specific information for lead technical personnel is contained in Chapters 5 through 9.

1.2. Handbook Organization

This section describes the organization of this handbook.

This handbook is organized as a series of chapters that provide information about the process of evaluating and selecting an Ada compiler. It raises questions that should be answered by those responsible for choosing compilers or reducing the risk of using a specific compiler.

1. **Introduction:** Includes the purpose of this handbook and provides help in its use.

2. **Common Questions:** Presents commonly asked questions, succinct answers, and, where needed, pointers to more detailed information. Topics include both the technical and pragmatic issues of compiler selection. This chapter can be used (among other purposes) as an executive summary for the rest of the handbook and to review particular points.

3. **Compiler Validation and Evaluation:** Distinguishes between validation and evaluation and provides an overview of the types of information that should be part of a general evaluation strategy.

4. **Practical Issues of Selecting an Ada Compiler:** Discusses the recommended steps for developing and executing a strategy for evaluating and selecting an Ada compiler for a particular application.

5. **Compile/Link-Time Issues:** Presents selection criteria based on the options, performance, capacity, and human factors of the compiler and linker.

6. **Execution-Time Issues:** Presents selection criteria based on the options, performance, and capacity of the code generated by the compiler and the run-time system provided by the compilation system.

7. **Support Tool Issues:** Presents selection criteria based on the program library system, linker/loader, debugger, and target simulator.

8. **Benchmarking Issues:** Presents the issues surrounding the use of test programs in order to obtain quantitative information about an Ada compilation system.

9. **Test Suites and Other Available Technology:** Provides an overview of the tools and technology available today to assist in the selection of Ada compilers.

The following appendices are also included:

A. **Test Suite Summaries:** Contains the categories of tests included in five major test suites as described in their documentation.

B. **Compiler Evaluation Points of Contact:** Contains short descriptions as well as names, addresses, and telephone numbers of professional organizations, U.S. government organizations, evaluation technology producers, and other Ada information sources.

C. **Accessing Network Information:** Contains scripts for accessing relevant information about Ada and evaluation technology on the ARPANET.

D. **Acronyms:** Defines acronyms that are used in the handbook.

1.3. Tips for Readers

This section highlights techniques for quick and efficient use of this handbook.

In addition to the question and answer approach of Chapter 2, several other techniques have been used to help the reader make maximum use of this handbook:

- **Definitions:** The terminology of evaluation and validation can be found primarily in Chapter 3. Those familiar with these concepts may not need to read this chapter carefully.

- **Summaries:** A summary begins each major section. Each summary is centered and italicized for easy identification.

- **Action plans:** Actions and strategies necessary to select an Ada compiler are given in Chapter 4.

- **Bold and bullets:** Major points are emphasized by using bold headings within bulleted lists. The major points are then followed by detailed discussions.

The information contained in this handbook has a short half-life. Examples are the descriptions of current evaluation technology in Chapter 9 and the list of points of contact in Appendix B. This handbook will be reissued periodically to present up-to-date information about Ada compiler selection. It is important that users have the most recent information as part of their decision-making process. Ada compiler technology, particularly in the realm of embedded systems, is advancing rapidly.

The following topics are not covered (or are given only limited coverage) in this handbook:

- details of Ada language features

- details of using the Ada language

- details of specific test suites and evaluation technology
- details of specific Ada compilation systems
- comparisons of different Ada compilation systems
- language administration and policy issues
- government procurement regulations and issues
- issues concerning the decision to adopt Ada for use on a project

Readers interested in these topics are referred to the Ada Information Clearinghouse (see Appendix B), to the *Ada Adoption Handbook: A Program Manager's Guide* [Foreman 87], and to other points of contact mentioned in the appendices.

2. Common Questions

Thoroughly evaluating Ada compilers is more difficult and costly than one might expect. However, an inadequate effort at this stage may be even more costly to the program in the long term because it may adversely affect the progress of the project for which the compiler was chosen. Questions about evaluating and selecting an Ada compiler generally fall into three categories: questions about procedure, questions about compiler technology, and questions about evaluation technology.

This handbook provides the information that is needed to make well-informed decisions about evaluating and selecting Ada compilers. Some typical questions are presented on the following pages. Where needed, pointers to supplemental information contained in other chapters of the handbook are provided.

2.1. Questions About Procedure

Question: What are some important sources of information to consult before beginning an Ada compiler selection process?

Answer: Before starting a selection process the user should be thoroughly familiar with some of the top-level issues of using Ada. For this, the reader is referred to the latest edition of the *Ada Adoption Handbook: A Program[B Manager's Guide* [Foreman 87]. The Ada Programming Support Environment (APSE) Evaluation and Validation (E&V) Team has produced a reference manual [Wright Research and Development Center 88a] and a guidebook [Wright Research and Development Center 88b]. The former gives a broad introduction of the issues and definitions of evaluating APSEs (including compilers) and the latter provides pointers to some of the existing E&V technology. An overview of the issues has recently been published in *IEEE Computer* [Ganapathi 89]. The AJPO's Information Clearinghouse is a good source of information. On the ARPANET, the info-ada bulletin digest is a source of anecdotal information. Finally, the references in this handbook provide a number of valuable sources of information.

See: Section 4.9 and Appendix B.

Question: How much time should be allocated for doing a thorough, independent evaluation of an Ada compiler?

Answer: The circumstances under which the evaluation takes place are the driving factors. In general, it will be much easier to evaluate a host-based compilation system than a cross-development system. It will be much easier to evaluate a compiler for a system with which the user is familiar than to evaluate a new system. It will be much easier to evaluate a compiler if there are experienced evaluators doing the job. It will be much easier if the compilers being evaluated are mature and stable than if they are new products consisting of new components. It will be much easier if compilers for only one target are being evaluated. The level of effort should also depend on the size of the program for which the compiler is needed. As a general rule, expect to take from one to six calendar months for a reasonably thorough first-time evaluation, depending on some of the factors mentioned above.

See: Section 4.5.

Question: If a compiler has already been specified for a program, is there any reason to complete a rigorous compiler evaluation process?

Answer: Yes. Even if a program is required to use a specific compiler (for example, standard military computers may have only a single Ada compiler) an evaluation should be performed. If the compiler fails to meet minimum requirements for the program, then alternative system designs, new compiler procurement, processor waivers, or language waivers must be explored. If there are functionality or performance deficiencies, then cost and schedule may have to be modified and workarounds need to be explored.

See: Section 3.2.

Question: What are the potential consequences of selecting a poor Ada compiler for a particular application?

Answer: The costs of a poor selection can be many times the cost of a thorough evaluation. A program should be cognizant of potential problems. For example, error-prone or inefficient runtime systems are particularly difficult to work around and require high quality and timely vendor support. Unsupported features of the Ada standard may make it impossible to meet functionality requirements. Performance degradation under loaded conditions may make it impossible to meet performance requirements. Changing compilers in the middle of a project will also prove costly.

See: Section 4.5.

Question: What high-level procedures should be used to evaluate and select Ada compilers?

Answer: The general process at the highest level is the same in all cases. First, the criteria must be established. Next, the tests must be gathered or written to test the compiler against the criteria. Finally, the results must be analyzed to determine whether the compiler meets the criteria. Within these broad parameters there are many alternate paths. The breadth and depth of the methods may vary from one selection procedure to another and from one project to another, depending on the application requirements.

See: Sections 3.2 and 4.1.

Question: How important is a compiler vendor after an acquisition is made?

Answer: Vendor support after acquisition is usually very important, especially at the present time. Relatively new compilers rarely work without problems, and the responsiveness of the vendor to solving these problems in a timely fashion can make or break a project. In some cases, the vendor may be called upon to tailor the compiler to application requirements to make it more responsive to project needs.

See: Sections 4.8 and 6.4.

Question: What are the reasons for and implications of porting code between compiler systems from different vendors and porting code between different targets?

Answer: First, an application may have to be ported from one processor to another if there is a compelling performance gain in the new hardware or if there is a requirement to run on several different targets. This would generally happen late in the product life cycle. Second, an application may have to be ported from one compiler for a given processor to another compiler for the same processor. This would generally happen early in the life cycle if a compiler were found to be deficient or vendor support to be lacking. Third, porting may be required when code is reused from another application. Finally, porting is required when two compilers are used in the development phase, one for initial development and unit testing on a host-based system and the second for the final target system. Changing vendors can be as disruptive as changing either host system or target system (while keeping the same vendor). This is due to the idiosyncratic nature of the total environment in which the compiler operates, as well as the implementation-dependent choices the vendor has made with respect to machine-dependent features. Installing, invoking the compilation tools, downloading, and debugging are all quite vendor-specific.

See: Section 4.7.

Question: What kinds of compiler deficiencies are likely to show up in the later stages of the development cycle? What will be the impact of these deficiencies on a large project?

Answer: Some deficiencies do not show up until the systems being developed reach a certain size. These *capacity* problems are often difficult or impossible to correct or work around. Some runtime system bugs do not manifest themselves until a system has been running for a long period of time. For example, if a runtime system does not correctly allocate and deallocate storage for exceptions, the bug may not be detected until a large number of exceptions have been processed. The impact on a project at this stage can be devastating, because there is rarely time left in the schedule to adjust. Risk reduction strategies for these situations would include early system testing and a strong working relationship with the compiler vendor.

See: Sections 5.3, 6.2.3, and 6.5.

Question: How many vendors are currently producing Ada compilers? How many compilers are available from these vendors? For which targets are cross-compilation systems available?

Answer: As of early 1989, there were nearly 50 Ada compiler vendors. The AJPO's validated compiler list contained over 230 compilers. Cross compilers are available for host-based systems as well as for an increasing number of bare targets.

See: Section 3.1.

Question: Can a fair comparison be made between Ada and other languages?

Answer: What can be compared is an implementation of Ada with an implementation of another language. To make the comparison a fair one, it must be understood that Ada does more consistency checking at both compile time and execution time than many other languages. The tradeoffs must be recognized and acknowledged.

See: Sections 3.2.6 and 9.6.

Question: Is the evaluation of Ada compilers substantively different from the evaluation of compilers for other languages?

Answer: In many ways it is the same. The prospective users must determine the time and space efficiency of the compiler and the code it generates, as well as the usability of the compiler. What makes Ada evaluation more formidable is the complexity of the language, the number of implementations, and the incorporation of executive functions such as memory management and scheduling into the runtime system. Furthermore, Ada's program library system, which provides separate compilation with full consistency checking, provides an evaluation dimension not relevant in many other languages.

See: Section 6.1.

2.2. Questions About Compiler Technology

Question: Are there substantial differences among validated Ada compilers?

Answer: Yes. The fitness of a specific Ada compiler for use in a particular application has very little to do with validation. Validation is a process which is meant to test whether an Ada implementation conforms to the language definition specified in ANSI/MIL-STD-1815A. Fitness for use has to do with compile-time and execution-time performance, capacity, and user options, as well as many issues related to the user interface, documentation, and support of the product. There are compilers that are more suitable for educational purposes, some that are more suitable for MIS applications, and some that are more suitable for real-time applications. As such, proper evaluation and selection is a necessity.

See: Sections 3.1 and 3.2.

Question: Do all Ada compilers employ the same algorithms to handle the semantics of the Ada programming language? What are the best sources of information about differences in implementations?

Answer: No. There is a great deal of variation permitted by the Ada language standard and a substantial performance impact on the users of Ada compilers. Implementors are required to provide an appendix to their Ada reference manuals detailing their differences. The ARTEWG (Ada Runtime Environment Working Group) is one source of information on implementation dependencies [Ada Runtime Environment Working Group 87]. There is also a new group that has been established under Working Group 9 (WG9) of the International Standards Organization (ISO) umbrella called the Uniformity Rapporteur Group (URG). While the Ada Rapporteur Group (ARG) deals with language interpretation and maintenance, the URG makes recommendations about how implementations should handle certain implementation features. At the very least, evaluators should be aware of the kinds of issues raised by these groups.

See: Section 5.5 and Appendix B.

Question: How much optimization can Ada compilers be expected to perform? Are there any disadvantages to a highly optimizing compiler?

Answer: The level of optimization provided by today's compilers varies greatly. Many simple optimizations are routinely performed by some compilers, but there are many complex optimizations that can improve performance. The Ada Compiler Evaluation Capability (ACEC) tests for 24 categories of optimizations. Depending on the compiler, optimization levels can be requested with pragmas or compiler switches or optimization may be provided automatically. While most users desire the most efficient code possible, highly optimized code may present its own new problems; for example, bugs may be introduced by the more complex optimizations. Also, the debugger may react differently to highly optimized source code and may itself contain bugs. For example, if an optimization keeps values in registers rather than storing them in memory, the symbolic debugger must recognize the correct values at every point in the program.

See: Section 6.2.3.

Question: What are "Chapter 13" features? Why can they be important to the selection of an Ada compiler?

Answer: The name of Chapter 13 in the *Reference Manual for the Ada Programming Language* (*RM*) is "Representation Clauses and Implementation-Dependent Features." It is concerned with representation of data, alignment of data in memory, packing of data in memory, low-level I/O, data conversions, interrupts, machine code, language interfaces, storage allocation, and other low-level issues. For many applications, these issues are irrelevant, but for embedded systems they are critical. For many years these features were believed to be "optional" features of the language. The most recent Ada Compiler Validation Capability (ACVC) test suite (Version 1.10, required for validation after 7/1/88) has a number of tests against requirements in Chapter 13. However, there is still not thorough coverage. The ARG is developing language commentaries explaining what aspects of Chapter 13 *must* be supported by every implementation, but it is unlikely that their work will be completed before the middle of 1989. Evaluators are advised to test for those features required by their application.

See: Section 5.1.4.

Question: What aspects of interrupt handling must be evaluated? Why are there few tests for interrupt handling in the currently available test suites?

Answer: Interrupt handling is important to most embedded system applications, but not to non-embedded systems. Interrupt latency and exit times, as well as the functionality available in the interrupt service routine, should be determined. Such tests are difficult to include in test suites because there are many options for handling interrupts and special hardware is usually required to implement interrupts and to measure the time required to process interrupts. Interrupt handling is dependent upon target architecture, compiler vendor, application, and programmer.

See: Section 6.8.

Question: If a compiler generates reliable code that runs fast, why should any time be devoted to evaluating other criteria for the compilation system?

Answer: The developer should be concerned not only with the final product, but also with the process of developing the final product and the process of maintaining the final product. For this reason the compile/link time, the support tools, and the human factors must be given appropriate weight. In addition, vendor support must be considered, particularly for embedded applications.

See: Sections 4.8, 6, and 7.

Question: Is code expansion (i.e., the average number of bytes of object code generated per line of source code) the best measure of the amount of space that will be required by an application?

Answer: No. While code expansion provides one indicator of eventual size, another important space consideration is how well the linker excludes code that is not used. A sophisticated linker should eliminate from the loaded code any parts of the runtime system that are not used, as well as any parts of packages (including subprograms and objects) that are exported but not used. While many of today's linkers do a good job of this, there are still some compilation systems that generate tens or even hundreds of thousands of bytes of code for the simplest of programs. Also important is whether the compiler shares bodies for generic instantiations and how much space is required for dynamically allocated objects.

See: Section 6.3.

Question: Is it sufficient to measure the compile/link time on a variety of large programs to determine the compile/link time efficiency of the compiler? If not, what other factors are important to consider?

Answer: What is more important than the size of the program is the rate at which the compile/link time grows with the size of the program. Some compilers may perform adequately for small programs, but degrade severely as program size increases. Conversely, the processing overhead for small programs may be great compared to that of large programs. It is important to know how compile/link time is influenced by unit size, number of variables, **with** and **use** clauses, subunits, generics, etc. Also important is the cost of recompilation and relinking in the face of small changes. For example, after changing a comment in a compiled unit, does it take just as much work to recompile the modified unit? Does the compilation system avoid recompiling units that do not need to be recompiled? What is the impact on compile time of hardware configuration (disk and memory space, network overhead, etc.) or the number of files in the library?

See: Sections 5 and 7.1.

2.3. Questions About Evaluation Technology

Question: Are evaluations of individual Ada compilers available, so that the work required to evaluate Ada compilers would be reduced? What test suites are available?

Answer: Currently, very few evaluations of specific products are publicly available. There are several reasons for this. First, they are obsolete almost from the moment they are published because compilers are constantly evolving and improving. Second, they must be carefully documented and defended from challenges from vendors with vested interests. Notable exceptions are the evaluations of Ada compilers based on the Ada Evaluation System (see Section 9.4) that the British Standards Institute is beginning to publish. The major general-purpose test suites that are currently available are the ACEC test suite, the AES test suite, and the PIWG test suite. Some application-specific tests are available in the Ada Software Repository.

See: Section 9.

Question: Is it safe to take at face value the numbers that are being reported by compiler vendors or independent groups trying to address performance issues?

Answer: No. Vendors put their products in the best possible light. For example, vendors naturally choose favorable parameters for compiling, linking, and loading their programs when those parameters are not explicitly specified or ruled out in the instructions for running the benchmarks. Independent groups, however well-intentioned, may not always control all the appropriate sources of variation nor provide sufficient documentation and disclaimers. Only by asking detailed questions about the configuration and the circumstances under which the numbers were derived can one start to gain confidence in what is measured. Even then, the questioner must be confident that the questions are being answered by informed and totally truthful sources.

See: Sections 8.2 and 8.3.

Question: What are the advantages and disadvantages of using a test suite that is available from a third party? What is the quality of these test suites? From whom are they available?

Answer: The test suites that are generally available are described in Chapter 9. The quality of these test suites is adequate if used by a knowledgeable tester who is familiar with most of the pitfalls of benchmarking. The advantage is that the coverage is very broad and detailed. (Almost every Ada feature is tested by these suites.) The suites are often helpful in answering very specific questions or in comparing compilers. Some have extensive tools for analysis of the results. They represent many work years of effort and have been constructed to avoid many of the common pitfalls. The disadvantage is that the tests do not necessarily test the features in the same combination or under the same circumstances in which they will be used by a particular application.

See: Sections 4.1, 9, and 9.1.

Question: What are the generic categories of test programs and what are the advantages and disadvantages of each?

Answer: One way that tests can be classified is by their granularity. There are fine-grained test programs that test and measure individual Ada features, such as the time and space for a subroutine call or the effect of a particular kind of optimization. On the other hand, there are coarse-grained tests that test many features in combination. The coarse-grained tests can be either synthetic (statistically constructed to look like a "typical" application) or actual application code. Fine-grained tests do not show how various features interact and are more subject to anomalous variations. Coarse-grained tests only give a broad indication of quality and do not pinpoint the source of difficulties. In the end, a combination of fine- and course-grained tests would seem to be advisable.

See: Section 8.1.

Question: Are benchmark test suites the best source of quantitative data for comparing compilation systems?

Answer: The best software to use for comparing compilation systems is the operational software itself. If operational software is not available during evaluation, then subsystems or prototypes can be helpful. In general the better one understands any software that is being used for evaluation (including test suites), the more meaningful will be the results.

See: Section 8.

Question: What hardware, aside from the host and target systems, is required to perform an assessment of Ada compilers?

Answer: For general purpose systems, no additional hardware is required. For embedded systems, there should be some hardware mechanism, such as a logic analyzer or in-circuit emulator, to verify software timings and to time those software segments, such as first-level interrupt handlers, that are not easily timed using software techniques.

See: Section 4.2.

Question: What software, aside from that provided by the compiler vendor, is required to perform an assessment of an Ada compiler?

Answer: For general purpose computing systems that are self-targeted, installing the compiler is usually straightforward. On cross-development systems it is often necessary to write some drivers for the target to adapt the runtime system and download software for the particular board that is installed. Typically, the adaptation of these target resident software modules is necessary to provide access to an on-board timer and a serial controller for downloading and host-target cross I/O. For either type of system, an assessment requires test software. It is often useful to have analysis software to reduce the large amounts of data generated by benchmark programs to useful information that can be easily interpreted.

See: Section 4.3.

Question: What are some of the most important criteria when selecting a compiler for hard real-time (deadline-driven) applications?

Answer: For hard real-time applications, some important criteria are usually deterministic behavior, the modeling of time, interrupt handling, representation clauses and implementation-dependent features, and time and space runtime efficiency. If the application will use the Ada tasking model, then the tasking features will be important criteria.

See: Sections 3.2, 5, and 6.

Question: What are some of the most important criteria when selecting a compiler for C^3I and MIS applications?

Answer: I/O performance and functionality, standardized interfaces to commercial off-the-shelf (COTS) software packages such as graphics, database, and windowing systems, and portability are likely to be rated as important criteria for these types of applications.

See: Sections 3.2, 5, 6, and 7.1.

3. Compiler Validation and Evaluation

There is an important difference between validation and evaluation. Validation tests conformance to ANSI/MIL-STD-1815A. Validation cannot be relied upon for determining whether a given Ada compiler is fit for use in a particular application.

3.1. Validation

Ada compiler validation is the formal testing process whereby an Ada compiler is certified by the Department of Defense to conform to the military standard ANSI/MIL-STD-1815A. This is accomplished by successfully executing several thousand test programs. Validation does not guarantee that the compiler (or the code it generates) is error free and it certainly does not imply anything about matters outside the realm of the standard (performance, capacity, and certain functionality areas).

This section describes what can be expected of a validated Ada compiler. Further, it describes what some have mistakenly come to expect of validated compilers. Finally, it presents what information can be gleaned from the validation process that is useful in determining the fitness of a compiler for use in a particular application.

The official procedures and guidelines for validation are contained in a 25-page document called *Ada Compiler Validation Procedures and Guidelines*, issued 1 January 1987 [United States Department of Defense 87c]. It is available electronically via the AdaIC Bulletin Board or the ARPANET. See Appendix B of this report for addresses and access information. Some changes to the validation process were announced on 25 October 1988 and are contained in a press release available from the Ada Information Clearinghouse. This announcement lengthened the amount of time that a validation suite is in force from 12 months to 18 months. It also lengthened the time that a validation certificate is valid from one year to the expiration date of the validation suite.

The Ada Compiler Validation Capability (ACVC) is a system used to test conformance to the standard. A part of the ACVC, the Ada Validation Suite (AVS) is a suite of test programs that are updated annually. The original philosophy and design of the testing capability was described by Goodenough [Goodenough 81, Goodenough 86a] in 1981. ACVC Release 1.10 was made operational on 1 June 1988. This version will be operational until 1 December 1989 when Release 1.11 will become the official version. Version 1.12 will be released for public comment and review on 1 June 1989, become operational on 1 December 1989, and will expire on 1 June 1991. Subsequent test suites will follow a similar release schedule.

Tests in the ACVC test suite are divided into three major categories and six subcategories [Goodenough 86a]:

- Non-executable

- Class B: errors to be detected at compile time
- Class L: errors to be detected at link time

- Executable
  - Class D: tests of the capacity of an implementation (need not be passed for validation)
  - Class A and C: other executable tests (the distinction between these two classes is no longer relevant)

- Other
  - Class E: tests whose PASS/FAIL criteria are special or that require special processing of some kind

The test names incorporate the class name so that it is clear from the name whether a test is expected to compile and execute. Executable tests are self-checking and print PASS/FAIL messages in a standard format.

The ACVC has been an effective mechanism for promoting conformance of implementations to the standard. Unfortunately, there has been a misconception that it should serve other purposes for which it was not designed. It is important to understand that validation was *never* intended to measure fitness for use. While the misconceptions surrounding the ACVC have largely been dispelled, it may be worthwhile to revisit them.

First, the ACVC is not a *guarantee* of conformance to the requirements of the standard. As Edsger Dijkstra has aptly pointed out, "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra 72]. Even though there are several thousand tests, most of the tests are small and few features are tested in combination with one another, so many potential problem areas are untested. Second, the ACVC does not systematically address capacity or performance issues. It contains a few tests for capacity and while the total running time for the tests is an indicator of performance, it is not sufficient for evaluation purposes. Third, the ACVC was not intended to address any issues of the programming support environment or human factors issues. A fourth misconception has been that the standard does not require a compiler to implement any of the representation clauses or implementation-dependent features described in Chapter 13 of the *RM*. Version 1.10 of the ACVC, in fact, includes many tests of Chapter 13 features, and more tests will be included in future versions of the ACVC. In particular, the ARG is developing language commentaries explaining what aspects of Chapter 13 *must* be supported by every implementation, but their work is not likely to be completed before the middle of 1989. Thorough coverage by the ACVC can therefore not be expected before Version 1.12. One rather subtle point regarding the ACVC testing is that the vendor is required to test the compiler using only one set of compiler options, when in fact the compiler may provide many sets of compiler options. The vendor is required to make no assertions about whether all

the tests are passed under all switch configurations. If there are a large number of such options, it may be unrealistic for the vendor to run all the tests in all configurations. The options and the interactions between options have the potential to introduce errors in the generated code. Three possible examples are optimization options, debugging options, and runtime configuration options (see Section 5.1).

In summary, the ACVC is a suite of tests that attempts to determine conformance to the language standard. It does not address, nor was it meant to address, evaluation issues such as fitness for use in any particular application area, performance, capacity, or availability of implementation-dependent features or options.

3.1.1. Validation Procedures

The current procedure for a compiler vendor is to obtain the test suite, indicate its intent to validate its compiler with one of the Ada Validation Facilities (AVFs) in the United States or Europe, negotiate a formal agreement for validation services with the AVF, submit a declaration of conformance and test results to the AVF, and resolve any test issues with the AVF. Following this, the AVF prepares a validation summary report (VSR), monitors on-site testing to duplicate previously submitted results, issues a validation certificate, and issues a final VSR. The director of the Ada Joint Program Office provides overall direction and is responsible for the Ada certification system. The Ada Validation Office (AVO) provides administrative and technical assistance to the director and the AVFs. A compiler that is validated according to the procedure outlined above is called a "base compiler." A "derived compiler" may be any of the following: a base compiler on an equivalent configuration (same computer architecture and operating system), a maintained compiler on a base configuration, or a maintained compiler on an equivalent configuration. For compilers that are derived from a validated base compiler, there is a registration procedure which conveys validation status without the completion of all of the validation steps. The status of a derived compiler as a validated compiler may be challenged, and if the challenge is sustained, the vendor must correct deficiencies within 90 days or the compiler will lose its validated status.

The list of validated Ada compilers dated 1 February 1989 contained 164 validated base compilers and 70 derived compilers from over 50 vendors. The current list, which is updated monthly, is available through the Ada Information Clearinghouse and electronically on the ARPANET. (See Appendix C for accessing information.) The list contains the vendor and compiler names, the host and target systems, the ACVC version number, and the expiration date for the validation. Over 50 of the compilers are targeted to processors that are bare targets or targets traditionally used in embedded systems. These targets include instruction set architectures by Intel, Motorola, National Semiconductor, and Data General, as well as several implementations of MIL-STD-1750A.

Validations eventually expire, so it is important to understand the status of a compiler being used on a project when that compiler is no longer validated (either because the project chooses not to upgrade to a newer version or because no subsequent compiler is available from the vendor). To address this issue, there is the concept of a "project-validated compiler." After a compiler has been baselined in accordance with applicable DoD policies on

software life-cycle management, it becomes a project-validated compiler for the lifetime of the project.

3.1.2. Validation Summary Reports

A validation summary report (VSR) is prepared by an AVF and contains results that are observed from testing a specific Ada compiler or grouping of Ada compilers. This is an important output of the validation process for users trying to evaluate Ada compilers. The VSR includes the following components [United States Department of Defense 87c]:

- Declaration of conformance.
- Description of all ACVC tests that were processed on the base compiler.
- Table showing the class and category of all ACVC tests and their results (e.g., total number of class C tests passed, failed, withdrawn, or inapplicable, etc.).
- Description of the testing environment (e.g., designation of configurations tested, testing completion date).

The VSR reflects any decisions made regarding disputed test issues. Finally, the VSR includes implementation-dependent options that must also be supplied in Appendix F of the vendor's reference manual.

The declaration of conformance is the certification by the implementor and owner of the compiler that they have implemented Ada as defined in the *RM* and that they have not deliberately included any extensions to the Ada language standard. This declaration must be submitted for the original base compiler, as well as for any derived compiler registration.

The implementation-dependent characteristics that must be included in the VSR and in Appendix F of the vendor's reference manual includes the following information:

- The form, allowed locations, and effect of every implementation-dependent pragma.
- The name and the type of every implementation-dependent attribute.
- The specification of the package SYSTEM (see *RM* 13.7).
- The list of all restrictions on representation clauses (see *RM* 13.1).
- The conventions used for any implementation-generated name denoting implementation-dependent components (see *RM* 13.4).
- The interpretation of expressions that appear in address clauses, including those for interrupts (see *RM* 13.5).
- Any restrictions on unchecked conversions (see *RM* 13.10.2).

- Any implementation-dependent characteristics of the input/output packages (see *RM* 14).

The validation summary report should be required reading for anyone selecting an Ada compiler for a project. The vendor should supply a copy upon request. Failing that, a copy can be obtained from the Ada Validation Office. (Unfortunately these reports are not online and the most recent reports may be difficult to obtain.)

It should be noted that there is currently little consistency among vendors in the quantity and quality of information provided in their versions of Appendix F. Application experts should examine them carefully to determine whether sufficient information is provided.

3.2. Evaluation

> *Ada compiler evaluation is the process whereby a user determines the fitness of an Ada compiler for use in a particular application environment. Evaluation is a much broader investigation than validation and includes factors such as the performance and capacity of the compiler and the generated code, the cost of the compiler, quality of documentation and error messages, vendor support, and quality and usability of the supporting tool set.*

This section describes some of the criteria that can be used to evaluate and eventually select an Ada compiler so that the reader gains a general appreciation for evaluation issues. Subsequent chapters give more detailed treatments of the evaluation issues. The criteria are both quantitative and qualitative in nature. While it may be appealing to use only criteria for which there is a numerical "score," it should be recognized that there are many criteria for which a simple number is not sufficient. While scoring may simplify and accelerate the process, the overall result may be inferior to one that includes qualitative information.

A comprehensive reference for criteria that may be used for evaluating software in general and Ada Programming Support Environments (APSEs) in particular is the *E&V Reference Manual* [Wright Research and Development Center 88a]. The following is taken from the executive summary:

> The purpose of the *E&V Reference Manual* is to provide information that will help users to: 1) gain an overall understanding of APSEs and approaches to their assessment, 2) find useful reference information (e.g., definitions) about specific elements and relationships between elements, and 3) find criteria and metrics for assessing tools and APSEs and techniques for performing such assessment. The latter are found (or referenced) in a companion document called the *E&V Guidebook* [Wright Research and Development Center 88b].

Some assessment criteria are more amenable to quantitative analysis while other criteria are more amenable to qualitative analysis. Examples of quantitative criteria that are addressed in this handbook include performance efficiency (both at compile time and at run time), capacity, and cost. Examples of qualitative criteria that are addressed in this handbook include correctness, completeness, and usability. The following sections address in general how each of these criteria may be evaluated.

3.2.1. Quantitative Criteria and Benchmarks

Quantitative criteria are distinguished by the fact that they can be easily measured with reliable, acceptable, and repeatable tests. Tests that measure the performance and efficiency characteristics of an Ada compiler are often called *benchmarks* because they are a means of easily comparing one system to another. An example of a performance criterion is compile-time efficiency. Tests may be written to measure the time it takes to compile various Ada programs under various conditions. A single test is not sufficient because one would like to know whether the compile time varies with the size of the program, with the Ada constructs used in the program, with the number of subunits used, with the compiler options, the machine configuration, etc.

Similarly, tests may be written to measure the time to link independently compiled modules and the time or space efficiency of the code generated by the compiler. It may be desirable to know how long it takes to execute an assignment statement, a subroutine call with three integer parameters, or some representative section of code. It may be important to know how much space is required for implementing these Ada statements or for certain data structures such as records. Capacity tests may be constructed to determine how large a program can be handled by an Ada compiler or how large a symbol table can be handled in a given system configuration.

3.2.2. Qualitative Criteria and Checklists

Qualitative criteria are distinguished by the fact that they are not easily measured with reliable, acceptable, and repeatable tests. There are always methods of quantifying such information, but the resulting numerical results are rarely more useful than anecdotal and summary information. Consider *correctness* as one example. There is no reliable method of determining how "error prone" a compiler is. Indicators include trouble reports to the vendor (rarely available), number of errors encountered running some series of test programs, or inspections of the compiler's source code (rarely accessible). However, there is no test for determining how many errors remain in any large software system and a compiler *is* a large software system.

Completeness refers to the extent to which a component provides the complete set of operations necessary to perform a function. If there were a master list of desirable functions and a weight associated with each function, then it would be simple to define a metric for the completeness of a particular aspect of a compiler. However, there is little agreement about the functions and how they should be presented. In an Ada compilation system, there are many implementation-dependent options and features. The standard says nothing about a requirement for a debugger or the features it should have. It says nothing about the error messages or their contents.

Usability is the effort required to learn, operate, prepare input for, and interpret output of a component. Certainly this criteria depends on the background and experience of the user. Ease of learning is often in conflict with ease of use. All human factors criteria are subjective because they involve human judgement and human preferences. For example, there will never be general agreement on the type of human interface that is best for displaying information on a bit-mapped workstation.

What all these qualitative criteria have in common is that they can be addressed with a series of questions or checklists that make it easier to evaluate them. Questions about debuggers include whether breakpoints can be set on subprogram entry and exit. Questions about error messages include whether they provide easy identification of the source of the errors. There is always a tradeoff between the ease of evaluating a simple yes/no questionnaire versus the more subjective questionnaire, but despite the problems, it is important to gather subjective and qualitative information.

3.2.3. Other Evaluation Techniques

Evaluators need not rely solely on information generated internally. Much information is available from third party sources. The validation information discussed in Section 3.1 is readily available. Vendors can be called upon to provide presentations about their products. These presentations must be received with the foreknowledge that vendors will always present their products in the most favorable light.

Perhaps the most useful source of third party information is available from other users of the product. No good evaluation should be considered complete without extensive interviews with current users. They have experienced firsthand the joy and pain of installing, testing, operating, and using the product.

Another source of information is an evaluation service. The British Standards Institute (BSI) has started a service to evaluate Ada compilation systems. As of early 1989, this service has conducted two compiler evaluations. Their methodology is a useful source of information and their evaluation reports are available for under $500. (See Section 9.4 and Appendix B.) The BSI plans to expand their Ada compiler evaluation service to the United States.

3.2.4. Reevaluation

There are three circumstances under which a project may wish to consider reevaluating a compilation system. These are the need to upgrade the compiler to a new version, to rehost, or to retarget. Each of these changes can generate a considerable amount of disruption for a project and should not be undertaken lightly. None of these circumstances, however, warrants a complete reevaluation and the cost in terms of time and schedule can be much less than that of an initial evaluation. Upgraded compilation systems suggest some reevaluation of the areas changed the most. A rehosted system suggests major emphasis on the compile-time testing. A retargeted system suggests major emphasis on the runtime testing and runtime system testing. It is probably a good strategy to keep the results of the initial testing so that a subset of the tests can be rerun and compared with the original results.

3.2.5. Tailoring Evaluations

It is highly unlikely that an evaluation of a compiler undertaken for one project will meet the requirements of another project. The many dimensions of compiler evaluation will almost invariably be weighted differently by different users with different application requirements. Projects will certainly have different views of the importance of cost, compile-time performance, execution-time performance, and configurability of the run time to their particular effort. This handbook gives many of the dimensions along which the compilation systems may be evaluated.

Table 3-1 gives an example of how the major concerns might differ for two extremely different applications, a hard real-time, embedded application (labeled "Embedded") and a non-real-time management information system application (labeled "MIS"). While the ratings of importance given in the table are arguable, it is clear that these two applications

Table 3-1: Application Concerns (Hypothetical Example)

Major Concerns of Application Developers		
Attribute	**Embedded**	**MIS**
Compilation speed	Important	Important
Efficiency of generated code	Very Important	Important
Efficiency of runtime code	Very Important	Important
Machine-dependent features	Very Important	Less Important
Execution restart/recovery	Very Important	Less Important
Text I/O functionality	Less Important	Very Important
Interfaces to COTS software	Less Important	Very Important
Portability of application code	Less Important	Important
Determinism	Very Important	Less Important
Timer resolution/accuracy	Very Important	Less Important
Support tools	Very Important	Very Important
Availability of runtime source code	Very Important	Less Important
Vendor support	Very Important	Important
Recompilation avoidance	Less Important	Important
Compiler correctness	Very Important	Very Important

have different requirements for an Ada compilation system and that what is optimum for one will not be optimum for the other. Other application areas such as soft real-time, C^3I, or educational applications would have still other major concerns. For example, educational application requirements would rank compile-time efficiency higher than either of the two applications given as examples in the table.

Furthermore, the application area may dictate certain criteria. Embedded applications may require extensive machine-dependent features, while MIS applications may not require any. Hard real-time applications (those whose correctness depends on meeting severe timing deadlines) may require highly precise timing capabilities, while soft real-time applications (those whose time requirements are not mission critical) may be less demanding. Large applications with hundreds of thousands of lines of code may be more concerned with compile-time efficiency and recompilation characteristics, while smaller applications may not need highly efficient compile-time performance.

Inevitably, users will tailor their programming styles to the characteristics of the Ada compiler. If **select** statements are inefficient, they will not use **select** statements. If exceptions are expensive even when they are not raised, then they will avoid exceptions. If dynamic memory allocation is expensive, then they will create internal development standards to allocate all data statically.

It is important to understand the tradeoffs between the application requirements and programming style on the one hand, and the capabilities of the Ada compilation system on the other. The evaluation process should consciously and overtly tailor the criteria to suit the application and programming style desired. Perhaps there will be no compilation system that meets all the requirements, but it is important for the requirements to dictate the compiler rather than the other way around. The steps recommended for conducting an evaluation and selection are given in Chapter 4.

3.2.6. Comparing Ada with Other Languages

It can be expected that first-time users of the Ada language will have to justify its use relative to languages for which the functionality, performance, and risks are better known. While this can be done, there are many caveats that must be observed. First, while two languages can be compared on a feature-by-feature basis, it is more productive to compare the implementations of two languages and recognize that no matter how elegant or functional a language may be, it is the actual language implementation that will determine the success or failure of a project. Some of the following guidance is also relevant for comparing Ada language implementations with respect to a given set of criteria, because the evaluator must recognize that Ada functions are often more robust or are carried out in the language rather than in the operating system.

Among the issues to be considered in fairly comparing Ada to other languages are the following:

- **Compile-time checking:** Ada provides consistency checking across separately compiled units, which is not available is some languages. The cost

of this checking must be compared to the cost of separate tools to do this in other languages or the additional integration time required in other languages.

- **Runtime checking:** Constraint checking for subscripts and other variables is done automatically in Ada. For fair comparisons, either the checking should be turned off in Ada or added to the tests in the other language.

- **Data representation:** The size of variables may be different by default in two different languages. Integers and floating point variables must be made the same for a fair comparison. (This is also true for two Ada implementations.)

- **Compiler and operating system options:** Rarely do default settings of various options make for fair comparisons. Experienced programmers must determine how to set the parameters so that each language provides comparable functionality (e.g., optimization levels) and ample resources (e.g., working set sizes).

Recent studies (e.g., [Byrne 88]) have demonstrated that the runtime performance of Ada compares favorably with other languages such as FORTRAN, C, and JOVIAL. For comparable benchmark tests, roughly as many run slower in Ada as run faster in Ada. Three areas that have been cited for increased speed in Ada are automatic in-lining of procedures, the ability to use block move operations for slices of arrays, and the ability to perform optimizations across separately compiled units because of information kept in the program library.

4. Practical Issues of Selecting an Ada Compiler

After a certain amount of information about evaluation has been gathered, the process of evaluation and eventually selection can start. An evaluation plan should be developed to determine the process and the products of the evaluation. The resources available for evaluation must be determined and set aside. Timetables must be established for the process.

The purpose of this chapter is to identify some of the pragmatic issues for evaluation and selection of an Ada compilation system. The first section identifies a general selection process. This process must be tailored to individual requirements. The intent is to provide a general structure that is not biased toward any particular host-target environment or any particular application area. The remaining sections list some of the practical issues that ought to be considered when planning for Ada compiler acquisition.

4.1. Selection Process

The following ten steps can be performed for any evaluation and selection process. Each step may consist of many substeps. However, it should be noted that eliminating candidate compilation systems may be much easier than confirming that a compilation system meets all or most of the criteria. This suggests a two-stage evaluation and selection process wherein the first stage quickly eliminates candidates from further consideration.

1. **Gather general evaluation information:** First, the evaluator needs to get a general understanding of the evaluation process. Reading this handbook will provide a start in this process, but some of the references should be consulted as well, and more detailed information should be collected about the compilers available and the evaluation technology available.

2. **Plan overall strategy, including budget, personnel, and timetables:** There should be an evaluation strategy based on the resources available. It is desirable to have a formal written strategy, but at the very least an informal strategy should be determined. If it is determined that insufficient resources or time has been allocated, then the budget or schedule should be revised. The overall strategy should, in particular, determine the level of effort to be expended on each of the following steps.

3. **Understand project requirements:** Criteria for compiler selection *cannot* be established without a firm understanding of the problem to be solved. This handbook can provide little help in this area because the domain-specific project requirements are so varied.

4. **Establish criteria based on the nature of the project:** The criteria for selection are technological and non-technological. Technological issues such as compile-time issues, execution-time issues, and support-tool issues are discussed in Chapters 5, 6, and 7. Some of the business criteria are discussed in Section 4.8 of this chapter. Two alternative styles of specifying criteria may be chosen. First, it is possible to specify absolute criteria, e.g., the compilation

system must support feature x or must not take more than y seconds to perform function z on a given system configuration. Second, it is possible to specify relative criteria, e.g., a rating scale for each criterion to facilitate the comparison of different compilation systems. The style of evaluation depends on the application. In fact, a mixture of these styles, with some absolute and some relative criteria, is possible.

5. **Plan tactics in three areas:** Information about the compilation system comes from benchmarks, checklists, interviews, and informal information gathering from vendors and users. In each case the evaluator must decide how much effort to devote and how to allocate that effort between already existing technology and hand-tailored technology. The plan is largely a decision about reuse and about the tradeoff between searching for tests that address the criteria of the previous step and developing unique tests. The former approach maybe desirable if the existing tests are comprehensive, suitable for the application domain, well cataloged, and well organized. The latter approach may be suitable if there are highly specialized criteria or if the existing technology is not well organized.

    - **Benchmarks:** The results from running test programs and test scenarios are the primary source of quantitative data. Benchmarks should be selected based on application requirements. These tests may be run and verified by the evaluator or may be accepted from another source. In the latter case the evaluator should consider the source of the information and the reliability of the information.

    - **Checklists:** The answers to detailed questions about the compilation system are the primary source of non-quantitative data. The questions should be constructed so as to allow the minimum amount of subjectiveness or judgement on the part of the evaluator. Simple yes/no questions have the advantage of limiting subjectivity, but have the disadvantage of restricting the amount of information conveyed. Checklists may be completed by the evaluator or some third party and the same caveat applies here as applies to benchmark data.

    - **Interviews and information gathering:** Interviews should be conducted with users of the product that is being evaluated and information should be gathered about the vendor of the compilation system. Unlike the other two categories of information, this must be done firsthand.

6. **Create or find an evaluation testbed:** An evaluation must take place in an environment where the product can be used. Reliance on paper evaluation is risky. The first preference is to create a testing environment on the site at which it will be used and conduct the testing with the people who will be using the compilation system. The second preference is to find another site (preferably other than the vendor's site) that can be used for a limited amount of time to test the product. Only as a last resort should the choice of a product be based on limited demonstrations. Several products should be tested in this environment so that side-by-side comparisons are possible.

7. **Perform the evaluations:** After the proper environment has been es-

tablished, the testing can begin. The benchmark programs can be run and the checklists completed. An evaluation log should be kept and any problems and observations made during the evaluation that are not explicitly addressed in the criteria should be documented.

8. **Analyze the results:** Once all the data have been collected, the information must be organized to facilitate a decision. Benchmark data must be organized so that unusual or anomalous behavior is not inadvertently missed. Checklist data should be organized so that information that discriminates among products stands out from information that does not discriminate among products. Formal or informal weighting of the data should be used to separate the critical criteria from the desirable criteria.

9. **Select, procure, install, and accept the compilation system:** Once all the information has been analyzed, it should be possible to make a decision. Procurement may be a highly variable process, depending on the organization that is making the procurement. Installation, if the compiler has not been evaluated in-house, may require some tailoring to the particular host-target environment. Finally, there may be some criteria that the user specifies to the vendor in order to accept the product after some trial period.

10. **Provide feedback to vendors:** The strengths and weaknesses of the compilation system should be communicated to the vendor. Not only will this repay a vendor who may have provided an evaluation copy, but it will also accelerate the progress toward production quality products for particular applications.

4.2. Hardware Requirements for Evaluation

The hardware that must be assembled for evaluation purposes includes a host system and a target system, with the necessary interconnections. Certain test equipment is also valuable for a credible evaluation.

The host and target hardware are often selected prior to compiler selection and are therefore not an issue. If an Ada development is only a small part of the total computational needs of a project, and there is already an established infrastructure for using a given host and its operating system, then the existing system is a natural and pragmatic choice for the additional Ada development effort. There is much to be said for continuing to work in a programming support environment that is familiar and comfortable. If, on the other hand, there is no such history, or if there is a desire for a revolutionary change, or if there is no Ada compiler for a given host, then other options must be explored. The time required to select a host system is therefore highly variable.

The choice of a target architecture is often determined by military standards. The Navy AN/UYK-43, AN/UYK-44, AN/AYK-14 computers and the Air Force MIL-STD-1750A architecture are examples of these standards. While some military programs require a particular architecture, other programs have no architecture requirements at all. When there is some discretion allowed, then the decision about the target system should be made with full

consideration of the Ada compilation systems currently available or likely to be available. The state of the compilation systems available for a given target architecture may be much more critical to the success of the project than the nuances of different instruction set architectures. Standard architecture benchmarks such as Whetstone may show one architecture to be slightly better than another, but the software may have a system impact far greater than the architecture.

For cross-development systems, the download path may be a key bottleneck. If the host is connected to the target by a 9600-baud serial link, the download time for a 64-kilobyte load image will be a little over a minute. For one-half a megabyte, the time will be nearly 10 minutes. Special hardware may be available for certain host-target combinations so that this transfer can be done at much faster rates. Also, this bottleneck may be mitigated by software that permits the sending of only that part of the download images that changes from one run to the next and by placing the receiver, debugger, or parts of the runtime system in ROM. The evaluator should carefully consider the impact of download times and investigate alternatives to serial transfer.

Test equipment can be a critical hardware resource for time-critical applications. A logic analyzer or microprocessor development system may provide critical timing information for real-time systems that is not easily available from software timing. Another advantage to hardware test equipment is that it is non-intrusive. The software will run exactly the same whether timing information is being collected or not. This equipment is particularly helpful for isolation of hardware faults and the measurement of interrupt latency times. For distributed applications it is important to have the capability to correlate the timing information from several independent sources.

4.3. Software Requirements for Evaluation

> *The software that is required for an evaluation is dependent on the hardware configuration. Normally, the host operating system will not be an additional item required. The target system, if different from the host, may require its own operating system. Software may also be required to download code from the host to the target. Timing software may be required that is more accurate than Ada's clock.*

The compilation system normally includes the following software components:

- compiler
- library manager
- runtime system
- linker
- downloader or loader

- debugger
- assembler
- simulator

The first five items are absolutely necessary to run Ada programs. The debugger is highly desirable, and the assembler and simulator may be desirable for certain embedded applications.

The environment in which the software runs on the host system, commonly called an Ada Programming Support Environment (APSE), may contain additional tools. These may include graphical design tools, static and dynamic analyzers, testing tools, pretty printers, etc. These are quite important and the integration of the tools is a critical issue for performance and productivity, but it is not within the scope of this report. However, it is important to compilation system evaluators to get a general feeling for the extent to which the tools listed above cooperate with the tools of the operating environment. For example, is the Ada library compatible with the file system of the operating environment and do all the commands of the operating environment apply to the Ada library?

The environment in which the software runs on the target system (in the case where the target is different from the host) may be supplied by the vendor (the so-called "bare target") or may be an operating system or an executive. In either case, the vendor must tailor the product to the target operating environment. Unfortunately, this is a burdensome task due to the nuances of different single board computers. Even for a given architecture there are different input/output and timer characteristics. Thus, the user may be faced with the task of tailoring part of the target environment. For example, the downloader may have to be tailored for a particular board or the timer interface routines of the runtime system may have to be rewritten. The user should be certain to specify precisely the target configuration so that the extent of the tailoring (if any) is known beforehand.

4.4. Test Suite Requirements

> *A test suite can be procured in a number of ways, depending on the test suite. The cost of obtaining the test suite is often negligible compared to the cost of setting up and running the test suite and correctly interpreting the results. Therefore, it is important to have specific test objectives in mind before starting out.*

The decisions regarding test suites are the following:

- Should a test suite or suites be acquired, or should a test suite be built?
- If a test suite or suites are to be acquired, which one or ones should be acquired?
- Of the test suite or suites acquired, should all or just some of the tests be run? If the latter, which ones?

The advantages and disadvantages of several test suites generally available are discussed in Chapter 9. Acquisition costs are generally very small and should not be a deterrent. On the other hand, the cost of setting up and using one of the test suites can be substantial. The questions above can only be answered by doing an evaluation of the test suites themselves, based on the level of effort and the criteria of the compiler selection process. If a short and unsophisticated evaluation is planned, the PIWG tests may be sufficient. If a more in-depth evaluation is planned, the ACEC or AES test suites may be more suitable.

4.5. Timetables, Dependencies, and Costs

> *It is difficult to provide any guidelines or rules of thumb for the schedule and budget required, in general, for an evaluation of a compilation system. Experienced and knowledgeable evaluators will require less time than novices. If the selection is dictated or there is only one compilation system for a military standard computer, the evaluation process may be significantly simplified. On the other hand, if there are many variables (host, target, and compilation system) and the risks of a wrong decision are high (as in the case of a major weapon system), then a substantial evaluation effort is called for.*

The time required for an evaluation is often dictated by the higher level considerations of a program schedule. Every attempt should be made to ensure that sufficient time and budget are allocated for the evaluation, but often the schedule must drive what can be accomplished. It is senseless to plan a six-month evaluation if only sixty days have been allocated to the job. The ten steps outlined in Section 4.1 should each be allocated a proportion of the time and budget, depending on the evaluation requirements. While these steps are presented as sequential, some in fact can be performed in parallel. The planning steps (steps 1 to 5) can for the most part proceed somewhat in parallel, although it is certainly desirable to know the strategic directions and criteria before planning the tactics. Steps 6 through 9 must be performed sequentially for the most part. Figure 4-1 shows a hypothetical schedule for a rather thorough first-time evaluation.

Figure 4-1: Hypothetical Milestone Chart

The schedule and budget for the planning process are highly dependent on the level of effort allocated for the evaluation process. As a general rule of thumb, it may be prudent to allow 25% of the total effort to the planning steps. Creating the testbed is highly dependent on what is already in place and the degree of variability permitted in the decision. If the host already exists, then little time is required to evaluate, select, procure, install, and check out the host. If the target already exists in the testbed, then this is also a zero cost item. Bringing an unknown target into a testbed and connecting it to an existing host system can take many months of effort. Acquiring and installing the test software for the testbed will depend greatly on the software chosen. At a minimum, the cost is nearly zero and the time is probably at least a couple of weeks. At the maximum, the cost is several thousand dollars and the time required may be more than a month, including setup and checkout.

Performing the evaluation also is highly variable. Very little of significance could be accomplished in under two weeks' effort. More realistic is a couple of months to run the tests, complete the checklists, and contact the vendors and users. Certainly something is to be learned about the test technology by executing the tests, but there are no significant gains after the second or third evaluation. Analysis of the results can be very time-consuming, but can be facilitated by appropriate software. For example, the ACEC suite has a good program for comparing the benchmark results from several compilation systems. This step depends on the level of overall effort, but a couple of weeks is probably again a minimum. Finally, if the data are well presented and analyzed, the selection takes minimal effort. However, the procurement, installation, checkout, and acceptance may take as much as several months.

One strategy that has proved effective in at least one case is to defer the final selection of a compilation system until top-level design is almost complete. Not only does this allow more time for compilers to mature, but it also provides time for some prototypes or project-specific tests to be developed and run on the candidate systems.

In summary, the scheduling and budgetary impact of compiler evaluation and selection is highly dependent on the situation. It can take from one to six months or more to complete with one to three people. The costs can vary from very little to tens of thousands of dollars when personnel costs are factored in. What is important to understand are the tradeoffs between the costs of doing a thorough evaluation and the long-term costs to a project of choosing an unacceptable, or less than desirable, compilation system.

4.6. Defining Requirements and Criteria

> *The requirements for evaluation and criteria for selection are extremely dependent upon the application. Requirements that are appropriate for a particular project must be developed by project personnel. This should be done with a balance between high-level system criteria and low-level technical criteria. There should also be a balance between development environment, runtime environment, and business criteria.*

This handbook does not attempt to define requirements and criteria for compiler evaluation and selection because every application should have its own. Categories of criteria can be specified safely, but it should be recognized that the depth and breadth of each category will be different for each project and different depending on the time and resources available for the evaluation and selection process. Chapters 5, 6, and 7, together with Section 4.8, should provide the consumer with the bulk of the criteria categories that should be covered by an evaluation. Chapter 9 gives pointers to some of the technology to help evaluate the various criteria with respect to particular compilation technology.

One real danger of this step of the process is the overspecification of requirements. Rarely will one know *a priori* all the functional, performance, and support requirements of a compilation system. Unless there is a known model of the system to be built, with accurate data on the loads to be put on the system, and known interactions between components of the system, it seems silly to get down to the level of specifying how long each feature of the Ada language should take at run time. It is better to base a selection on a high-level criterion rather than a low-level criterion because the interaction of various features is unknown. For example, it would be better to specify for an MIS application that a certain file-processing program execute in a certain amount of time rather than to specify the requirements for each atomic action, such as accessing each record of the file.

It is unlikely that all the criteria for selection will receive equal weight in an evaluation. Those who are quantitatively minded will want to score each criterion for each candidate compilation system (on a scale of 0 to 100, for example,) and then weight each criterion (as a percentage of unity). Then when all the weighted scores are added, the score for the compilation system will also be based on a scale of 0 to 100. For those who are subjectively minded, the process can be more informal, with a subjective rating being given to each compilation system based on all the criteria. There is no single correct way to perform a rating of products and none should be imposed.

4.7. Portability Issues

> *In the selection of an Ada compiler, one portability issue is whether the application code can be ported to another host or target system. This might happen when a system is modernized. Another issue is whether the existing code can be ported to another compilation system for the same target.*

In the lifetime of some programs, it may be more likely that there will be a need to change a

compilation system (for a given target system) than there will be to change a host or target machine. Many machine upgrades fall into the category of "upward compatible" upgrades in which a processor is replaced with one that is faster or one that has an extended or reduced instruction set. The reason that there is some stability in instruction set architecture (ISA) is that so much depends on the ISA that a change in the ISA would mean changing almost every other aspect of the environment. This is true of both a host system and a target system.

A change of the compilation system, on the other hand, can be undertaken without changing the entire supporting environment. A change of compilation system may be required for a number of reasons including:

- requirement for additional performance
- requirement for additional functionality
- lack of support from the vendor
- lack of development activity on the part of the vendor

If the vendor ceases operation, then the user faces the prospect of maintaining the compilation system without vendor support or porting to a new compilation system.

It must be recognized by the consumer that highly tailored compilers may be the only way to satisfy performance constraints, but that each special feature that is included may detract from the portability to a new system. Thus, the advice for the consumer is to understand whether the performance required can be achieved with a given compilation system without using implementation-dependent features. If it cannot, then the user must be aware of the portability constraints, isolate the implementation dependencies into a few small packages, and carefully document them so that any port would be made easier.

4.8. Evaluating Vendors

> *The guidelines for evaluating vendors of Ada compilation systems are not very different from the guidelines used to evaluate vendors of other hardware and software products. There should be some reasonable assurance that the company is financially healthy and that it will be able to service and support the product it is selling. Responsiveness of the vendor to error reports as well as to requests for tailoring the compiler should be consistent with the needs of the project.*

The following issues are important for the evaluation of an Ada compilation system vendor:

- **Corporate structure:** Is the developer of the product the same as the distributor of the product? If not, what is the relationship between the two companies? Is the distributor knowledgeable about the product? Is the entire compilation system produced by one company? If not, how are problems reported and fixed?

- **Corporate performance:** Has the vendor produced product releases on schedule? Is the vendor responsive to requests for information? Does the vendor provide an appropriate customer interface?

- **Product lines:** How important are Ada compilation systems to the company's overall business? Does the company specialize in a particular hardware domain or provide rehostable and retargetable compilation systems?

- **Corporate health:** What is the *primary* business of the vendor? How long has the company been in business? Is it profitable? Is the number of employees increasing or decreasing? How many employees are working on technical development? On supporting the product?

- **Tailoring policies:** Is the vendor willing and able to tailor the compiler for specific application requirements? If so, what will be the cost and schedule? How do the changes affect the maintenance agreement?

- **Support policies:** Is there local product support? Is there a telephone hotline? What is the escalation policy for problems? How are problems reported and tracked? What are maintenance response times? Are bugs fixed? How often are there new releases? Is there an online database of reported problems? Is it available to customers? Is there a product newsletter? Are there user groups? Are there electronic bulletin boards for customers? Can support personnel be contacted by electronic mail systems? Are previous versions of the product supported? What does the maintenance provide? Are there any response guarantees for reported errors?

- **Pricing policies:** How much does the product cost? How does the price depend on the characteristics of the host? The target? Are discounts available for quantity purchases? What is the cost of maintenance?

- **Runtime royalties:** Does the vendor charge a royalty for each copy of an application program (with the vendor's runtime system) that is deployed on a separate target system? If so, what are the procedures for accounting for and collecting this royalty? **Note:** This royalty could be by far the largest component of cost for applications that are duplicated in thousands of systems (such as many weapon systems). It is important to understand the implications of these royalties *before* selecting a vendor.

- **Source code:** Is source code available for the compiler and runtime system? If so, what are the cost and licensing terms? If not, can the source code be put in escrow so that if the company goes out of business, the application system developer has recourse to solve problems and fix bugs?

- **Contractual issues:** Can the product be purchased or only leased? If it cannot be purchased, is the license perpetual or for a fixed term? If fixed term, what are the renewal terms? What happens in the event that maintenance has been dropped?

- **References:** Has the compilation system been used to develop software systems similar to the applications being considered by the buyer? If so, is the vendor willing to provide several references? **Note:** The buyer should insist on talking to technical references, not just project managers.

It should be noted that the AES contains an extensive "vendor/implementor questionnaire" that could be used for guidance in extracting important information about the vendor.

Vendors often emphasize future improvements to their products rather than their products as they currently exist. Evaluators should be skeptical of promises and try to place more weight on past performance. Significant improvements are difficult to achieve.

4.9. Getting More Information

The appendices and the reference section of this handbook provide sources of additional information. The most relevant and current information available in published form is contained in two newsletters:

- **Ada Information Clearinghouse Newsletter:** Published roughly four times a year by the Ada Information Clearinghouse for the Ada Joint Program Office, this newsletter presents information on the AJPO, Ada usage, validated compilers, and Ada policy and events.

- **Ada-JOVIAL Newsletter:** Published quarterly by the Language Control Facility at Wright Patterson AFB for the Ada Joint Users Group (AdaJUG), this newsletter provides articles, news, and announcements about both Ada and JOVIAL tools and compilers. It contains up-to-date information about Ada compilers and vendor points of contact.

Refer to Appendix B for addresses and contact information for the newsletters. For online information that may be even more up-to-date than what is contained in the newsletters, refer to Appendix C.

5. Compile/Link-Time Issues

> *The time and space efficiency of the generated code on the target system is often the primary criteria for Ada compiler selection. But there are other important criteria as well. The functionality and performance of the compiler and linker are selection criteria that cannot be overlooked.*

The purpose of this chapter is to raise the compile/link-time issues of importance to a compiler buyer. Each of the issues is described in a general way and some of the major criteria are identified. An exhaustive list of possible criteria is not given since such a list is highly dependent on the application. Rather, the reader is given an appreciation for the issues and provided references to more detailed information. Some compilers perform some linking operations while others do not. Further information about linkers and loaders is contained in Section 7.2.

5.1. Compiler Options and Special Features

> *Compiler vendors are free to provide a number of features and options not required by the standard. These include compiler switches, pragmas, attributes, and other machine-dependent characteristics. All implementation-dependent features must appear in Appendix F of a vendor's reference manual.*

5.1.1. Compiler Options

The ANSI/MIL-STD-1815A defines the syntax and semantics of the Ada language, but the compiler vendor is given latitude in providing options or features that are not required by the standard. There are two primary ways that the user can specify directions to the compiler. The first is through a directive to the programming environment when the compiler is invoked and the second is through a language construct called a *pragma*. It is possible that compiler directives and pragmas may give conflicting information. It is therefore important to consult the compiler documentation to determine which is the overriding direction.

The following list gives some of the compiler options that are often provided as directives when the compiler is invoked:

- generation of source code and machine code listings
- generation of machine code output
- specification of program libraries to search for input
- control of the printing, selection, and disposition of diagnostics
- control of the level of debugging requested
- control of the level of optimization requested

- control of ability to suppress runtime checks
- control of conditional compilation
- ability to terminate after syntax checking
- ability to terminate after some predefined error limit
- ability to print timing information about the compilation

5.1.2. Pragmas

Pragmas are used to convey information to the compiler. A pragma starts with the reserved word **pragma** followed by an identifier that is the name of the pragma and optionally by parameters. There are two kinds of language pragmas: those that are predefined in the *RM* and those that are defined by the compiler vendor. There are 14 predefined pragmas defined in Annex B of the *RM*. While it is true that *RM* 2.8(7) states "pragmas defined by the language ... must be supported by every implementation," the fact is that the level of support varies from implementation to implementation. For example, implementations differ in the languages to which they interface, the number of priority levels they support, and whether CONTROLLED or SHARED have any effect at all. In most cases, the validation suite verifies only that the pragma is *recognized* by the compiler, not that it has any effect. Therefore, it is extremely important in an evaluation effort to determine what pragmas are important. For example, certain pragmas (such as INLINE) may be particularly important to particular design approaches (such as object oriented design). The user documentation should be checked to verify the level of support provided for all predefined pragmas. It is highly desirable for the implementation to follow the recommendation of *RM* 2.8(11) that "implementations issue warnings for pragmas that are not recognized and therefore ignored."

Pragmas that are defined by the vendor are provided by the implementations to provide additional performance, to make the job of the implementor easier, or to allow use of additional functionality of the operating environment. For the sake of portability, a compiler may ignore a pragma if it is not recognized. Implementation-defined pragmas must be described in Appendix F of the implementor's reference manual. The existence of implementation-defined pragmas is becoming increasingly more important for improving performance of compilers and should be considered very carefully in evaluation efforts. However, these pragmas are not generally portable, and they cannot be tested using standard benchmark test suites.

The following list gives some of the areas in which compiler vendors provide implementation-defined pragmas:

- provision of interfaces to operating system service calls
- provision of interfaces to Ada objects and routines from other languages (import/export)

- choice of representation for predefined types in package SYSTEM
- ability to suppress all runtime checks
- specification of task scheduling discipline (e.g., time slicing)
- control of storage allocation for tasks
- ability to specify restricted use of interrupt service routines to permit fast interrupt handling
- ability to share code for generic bodies under certain conditions
- machine-dependent specification of register conventions and calling conventions

It should be noted that vendors provide functionality in different ways and that what one vendor provides by a language pragma, another vendor might supply as a compiler option or a customized package.

5.1.3. Attributes

Attributes are basic operations applied to an entity given by a prefix. Like pragmas, there are a number of attributes defined by the language standard (Annex A). Unlike pragmas, attributes must have the effect described in the *RM*. Furthermore, the implementation may provide implementation-defined attributes as long as the attribute designator is not the same as any language-defined attribute. The implementation-defined attributes must be described in Appendix F of the implementor's reference manual. Implementation-dependent attributes are much less prevalent than implementation-dependent pragmas and primarily address the problem of extracting, during run time, more information about the machine representation of Ada objects.

5.1.4. Other Important Compiler Features

There are a number of other important compiler characteristics that should be evaluated by a prospective user. It is not within the scope of this handbook to treat them in detail. Many are only of interest for embedded real-time systems. It is important that the vendor provide comprehensive documentation on the areas of interest for a particular application. The following list provides some of the areas in which there is a degree of variability from compiler to compiler. All the information about these areas is required to appear in Appendix F of the implementor's reference manual.

- **Predefined language environment:** Package STANDARD (defined in Appendix C of the *RM*) contains all predefined identifiers of the language and any implementation-defined types such as SHORT_INTEGER, LONG_INTEGER, SHORT_FLOAT, and LONG_FLOAT. (Note that some implementations may not use these names as their predefined types, although they ought to.)

- **Specification of the package SYSTEM:** This Ada package contains definitions of certain configuration-dependent characteristics, such as the sizes of integers and floating point numbers and the resolution of the clock.

- **Restrictions on representation clauses:** Ada defines length, enumeration, record representation, and address representation clauses. The extent to which an implementation must support these features is currently changing as the validation suite is being upgraded.

- **Conventions used for implementation-generated names denoting implementation components in record representation clauses:** *RM* 13.4(8) allows the implementation to define these conventions.

- **Interpretation of expressions appearing in address clauses:** *RM* 13.5(3) permits latitude for interpreting a value of type ADDRESS.

- **Restrictions on unchecked conversions:** Implementations may place restrictions on sizes or types of objects to be converted.

- **Implementation-dependent characteristics of input/output packages:** Some implementations aimed at specific application areas such as MIS may provide more sophisticated I/O packages which still have implementation-dependent characteristics.

-

- **Low-level input/output:** *RM* 14.6 defines the requirements for low-level input/output. Since the kinds and formats of the control information will depend on the physical characteristics of the machine and the device, the parameter types of the procedures are implementation-defined. Some implementations may provide alternative packages that interface with operating system services for this purpose.

- **Machine code insertions:** *RM* 13.8 defines a mechanism for inserting machine code in an Ada program using a package MACHINE_CODE. An implementation is not required to provide this package.

5.2. Compile/Link-Time Performance

The amount of time and disk space required to compile and link Ada programs is important for extremely large Ada projects. Many compile/link-time performance issues cannot be simply resolved by buying a larger computer. The distinction between compiling and linking in Ada is often blurred because some vendors postpone some operations such as generic instantiation until link time while others perform link operations at compile time.

The amount of time (and space) that it takes to compile an Ada program is of paramount importance during the development phase and may be less important during the maintenance phase. As second and third generation compilers appear, expectations have been

raised for the performance of production quality compilers. Unfortunately, the evaluation standards that are used are unsatisfactory for comparing performance. Typically, compiler vendors will quote figures giving the speed of a compiler in terms of lines of code compiled per minute. Very rarely do they quote the requirements for disk and memory for the compilation.

Unfortunately, the "lines of code per minute" metric is not well defined and may vary greatly depending on the definition and the programs being compiled. In fact, Firesmith [Firesmith 88] has shown that the number of lines in an Ada program varies by a factor of six, depending on the definition of a line of code. Among the factors influencing the "lines of code per minute" metric are the following:

- The machine and operating system (this should always be stated).
- The definition of a line of code (usually defined as the number of carriage returns or the number of Ada statements).
- The size of program (small programs may compile more slowly and generate more code per line).
- The number of comments and blank lines (if carriage returns are used for lines).
- The type of statements (wide variation exists in the size of Ada components and the difficulty of compiling those components).
- The compiler options selected (optimizations and debugging options generally increase compile time).
- The number of subunits and generics (these, in particular, may have a great impact on compiler performance).
- The definition of time (wall clock time includes I/O waits while CPU time does not).
- The operating system parameters (e.g., working set size).
- The size and state of the program library (how fragmented).

Vendor claims about compiler performance should be viewed with circumspection. Even if a vendor provides all the information listed above, it is difficult to evaluate unless compared to other compilers under exactly the same conditions. The evaluator is strongly advised of the need for controlled experimentation. Instead of being interested in a single number for time characteristics, the evaluator should be interested in how the time characteristics vary with respect to the parameters listed above. Furthermore, the programs that are used to discover these relationships should be typical of the programs that will be developed using the compilation system.

Among the questions to be answered are the following:

- How does compilation time vary with size of program, complexity of program, unit dependencies, and optimizations selected?
- Can the compiler be invoked simultaneously by more than one user?
- How does compilation time vary with system load (number of users)?
- How much disk and memory space is required for a minimal compilation?
- How much do disk and memory space requirements vary with the size of program, complexity of program, and other factors?
- Does the compilation system clean up after itself with respect to temporary files on disk?
- How much space is required for all the file objects derived from the source file during compilation and to what extent can these derived objects be controlled in size?
- Are there both batch and interactive modes and if so, how do they differ?
- Are there provisions to avoid unnecessary compilation or for incremental compilation (see Section 7.1.1)?

Because implementors can partition the effort of pre-runtime activities differently among the compiler, library manager, linker, and loader it is important for evaluators to understand and compensate for the tradeoffs. If improved compiler performance is gained at the expense of greater link times or vice versa, this should be made apparent in the evaluation results.

5.3. Compiler Capacity and Limitations

> *It is possible that some validated Ada compiler can compile only small programs. (In fact, some Ada compilers have been "crippled" so that they may be sold for a lower price to educational institutions.) For large projects it is important to know what the limitations of a compiler are in terms of number of statements, units, and identifiers, as well as the maximum size of critical data structures.*

The Uniformity Rapporteur Group (URG) of ISO WG9 has begun to define some standards to which Ada implementations ought to adhere, including minimum capacities that ought to be provided by Ada compilers. What is desired is that none of the capacities are unreasonably small so as to hamper software development. While it is reasonable for vendors to sell restricted versions of compilers for educational purposes, it is not reasonable to provide highly restrictive versions for general use. The following list represents some of the more important implementation limits that the user should find documented in the vendor

documentation. In the absence of user documentation, the Ada Evaluation System (see Section 9.4) contains a battery of tests for reasonable limits on most of the entities listed below.

- lines in a compilation or compilation unit
- compilation units in a compile
- **with**ed or **use**d units in a compilation unit
- packages, subprograms, and subunits in a compilation unit
- characters in an identifier
- identifiers in a program
- entries in a task
- static nesting depth for subprograms, loops, blocks, packages, subunits, **accept** statements, **case** statements, generics, **if** statements, and aggregates
- static nesting depth of parentheses in expressions
- characters in a line
- dimensions and elements in an array
- elements in an aggregate
- formal parameters in an entry, subprogram, or generic declaration
- enumeration literals in an enumeration type definition
- alternatives in a **case** statement or **select** statement
- **elsif** parts in an **if** statement
- characters in a string
- bits in an object
- discriminants in a record type
- number of instantiations of a generic subprogram or package

5.4. Human Factors

> *The productivity of the user of a compiler is strongly influenced by the user interface provided by the compiler. This includes how the compiler and associated tools are invoked, the quality of the error messages provided, the manner in which the compiler responds to errors, and the quality of the documentation. These qualitative issues can have a greater influence on productivity than quantitative measures, such as compile time, do.*

Human factors are often described in terms of ease of learning and ease of use. In order for a system to be easy to learn, it should be similar to other systems that are already learned. For example, if one is an expert using the DEC VAX/VMS operating system, then a compilation system that uses the same mechanisms to invoke tools and specify parameters will be easier to learn than one that uses UNIX conventions. The same is true of the style of the debugger, the file management tools, the diagnostics, and the user documentation. In order for a system to be easy to use, it must be helpful in uncovering errors early in the development process and provide succinct, but accurate information about the source of errors. The following sections list some of the areas that must be considered in evaluating the user interface of the compilation system.

5.4.1. Informational Outputs and Diagnostics

The compilation system provides information about the compilation as well as information about errors.

- **Compiler listing:** Is a cross reference listing giving point of definition and all uses available for all identifiers? Can the origin of **with**ed objects be determined? Are all options in effect clearly listed? Is the version of the compiler available on the listing? Is the total size of the code and data given?

- **Assembler listing:** Is it available? Is it interleaved with the source? Is it clear and concise? Can a mapping of the data objects and subprograms be obtained? Is linking information shown?

- **Error reporting:** How many severity levels are there for error messages? How accurate are the error messages at identifying the source of the error? How helpful are the error messages in fixing the error? Can more information be obtained about the error either from the documentation or interactively? Are error messages keyed to the *RM*?

- **Warnings:** Does the compiler provide warnings for implementation-dependent features and non-portable code? Does it warn about potential runtime errors such as infinite loops, uninitialized variables, and unreachable code? Are pragmas that have no effect flagged? Are unassigned **out** parameters, unreferenced **in** parameters, and functions without a terminating **return** flagged?

- **Interactive help:** Can a user get help on the use of the language and the compiler interactively? Is the *RM* online?

5.4.2. Error Recovery

The maximum amount of information should be derived from each compilation. A user becomes very frustrated if only one error can be detected in each compilation. The purpose of error recovery at the compile phase is to continue in the face of errors to the maximum extent possible and recover the context so that errors are not propagated through the compilation. Systems might recommend corrections to errors or even try to correct errors when directed to do so.

- **Error recovery:** Does the compiler terminate analysis of errors upon encountering difficult errors? Is there any attempt to correct simple errors such as misspelled keywords? Does a single syntax error prevent semantic analysis? Does a single error often cause a cascading of subsequent errors?

- **Handling multiple compilation units:** Does the compiler continue a compilation if errors are found in a previous compilation unit? Are all legal compilation units added to the program library?

5.4.3. Documentation

Good hardcopy and interactive documentation is indispensable for ease of use and ease of learning. Some characteristics of good documentation follow:

- **Contents:** The documentation set should include a definition of the functionality of the Ada compilation system, a user's guide that describes how to use the system and the informational outputs, and a description of the runtime system and its characteristics.

- **Organization:** There should be tables of contents and indices for each document supplied. A master index is useful when there are multiple volumes.

- **Style:** The documentation set should be clear, concise, complete, and written in plain English.

- **Implementation dependencies:** The documents should use color, changebars, or some other technique to clearly distinguish those features that are implementation-dependent.

- **Appendix F:** After the index, this is the most important part of the documentation. It should be complete and in the order specified in the *RM*.

- **Capacities and limitations:** Information about the compile-time and execution-time capacities and limitations should be clearly stated.

- **Implementation options:** The documents should give as much non-proprietary information as possible on the internal structure of the compiler and runtime system and the choices made by the compiler vendor to enhance functionality and performance. Performance characteristics of critical runtime operations as a function of the variables that influence timing should also be provided.

- **Error messages:** There should be a clear explanation of each error message reported by the compilation system with possible remedies.

- **Installation instructions:** The documentation should have installation instructions with recommended system parameters for users and sysgen parameters for the system.

5.5. Implementation Options

> *Many algorithms exist for accomplishing the requirements specified in the RM. For example, the method of choosing which task to run from a group of ready tasks is determined by the implementor. The actual algorithms are often difficult to determine by testing. The evaluator should attempt to learn how certain critical operations are carried out in a particular compiler. This information should be provided in vendor documentation, but often is not.*

There are many instances where the algorithm for specifying how to implement a given Ada function is left to the discretion of the implementor. Examples include which data structures to use for composite data types, whether generic bodies use shared code or not, or how the **case** statement is implemented. The URG of ISO WG9 is compiling a list of implementation options along with current practice. Such a list provides the evaluator with a baseline for comparing systems.

Some of the choices have little effect on the performance of the Ada compiler either at compile time or at run time. Other choices may have significant impacts. Sometimes the vendors treat this information as proprietary since they may feel that a certain technique gives them a competitive advantage. Certainly a highly optimizing compiler is valuable for real-time applications. The job for an evaluator is to determine those features that are critical to the application and then pressure the vendor to provide the information that will facilitate making an informed decision.

Some of the important compile-time, implementation-dependent issues being considered by the URG include:

- minimum requirements for arithmetic types
- minimum source line lengths
- file name conventions
- recommendations for pragmas SHARED, SUPPRESS, INTERFACE

The URG issues are most important to those applications that either have specialized or unusual requirements or that will be ported to other compilation systems.

6. Execution-Time Issues

The time and space efficiency of the code generated by the Ada compiler as well as the time and space efficiency of the runtime system must be carefully evaluated. Other considerations are runtime system capacity and implementation-dependent functionality provided at run time.

6.1. The Runtime System Model

Three execution-time models can be defined for Ada programs. First, there is a "host-based" environment in which the Ada code can make use of operating system services. Second, there is a "bare machine" environment in which all the execution-time software is provided by the compilation system. Third, there is an "enhanced bare machine environment" in which some of the execution-time services may be provided by a third party "executive." The issues are: how well does the runtime system code match user requirements and how well prepared are vendors of compilation systems to provide executive system functionality and performance?

Two components must be considered in an evaluation of the runtime performance of an Ada compilation system: the code that is generated from the application program and the code that provides the environment in which the application program runs. The latter is called "the Ada runtime system" and is provided by the Ada compilation system. The Ada runtime system provides the resource management for the Ada program. Among other functions, it must provide the following services: memory management, task management, time management, exception management, and I/O management. These services may be many times more costly in performance (time and space) than the code generated by the application program.

Further complication is that there is no clear-cut distinction made by implementors as to what functions cause code to be generated and what functions cause runtime system calls. For example, string concatenation can be done by in-line code or by a runtime system call. These decisions impact the speed of the operation as well as the size of the runtime. There are obvious tradeoffs between the size of the runtime with the size and efficiency of the generated code. Therefore, it is important to distinguish in any evaluation between the generated code and the runtime system and the tradeoffs involved.

Ada compilation systems whose generated code runs on the host development system ("host-based compilers") usually make heavy use of the services provided by the underlying operating system. In these cases, the runtime performance of the Ada system is unalterably tied to the performance of the underlying operating system and the manner in which the Ada system interfaces with that operating system. The compiler vendor generally has no choice but to use the underlying operating system for these services and the evaluator usually has no choice but to evaluate the compilation system in the context of the operating system.

Ada compilation systems whose generated code runs on targets different from the host development system ("cross compilers") have two alternatives. Either the Ada compilation

system can provide the entire runtime system or the runtime system can be provided to run on top of an executive. In either case, the full semantics of the Ada language must be provided. In the first case (a "bare machine environment"), the runtime system must be tailored to a specific computer because of differences in memory, clocks, and interrupt structures. In the second case (an "enhanced bare machine environment"), the runtime system must be tailored to the interfaces of the executive which, in turn, is tailored to a specific computer system.

Some of the components that vendors distinguish are the following:

- executive or kernel
- memory management
- tasking management
- exception management
- interrupt management
- predefined library packages (I/O, SYSTEM, CALENDAR, etc.)
- vendor-supplied library packages (math, bindings for COTS databases and graphics, etc.)

The advantages of a bare machine implementation are that the vendor has complete control and can provide a highly optimized system. The disadvantages of a bare machine are that the compiler vendor may have less experience in performing some of the executive functions than an executive system vendor and therefore have a less mature product. For a more complete exposition of the concept of a runtime environment and how this concept differs in Ada as compared to other languages, the reader is referred to an article entitled "A Framework for Describing Ada Runtime Environments" [Ada Runtime Environment Working Group 88a]. For information on selecting, configuring, and using an Ada runtime system, as well as checklists and evaluations of specific Ada runtime systems, the reader is referred to [U.S. Army HQ Center for Software Engineering 89]. Figure 6-1 gives a very simplified view of the three models of Ada runtime environments.

Small, high performance, real-time executives have been in existence for a number of years, both as commercial products and as company proprietary products. Ada runtime systems for bare targets have been available for only a few years. Providers of real-time executives are extremely cognizant of time-critical constraints of interrupt handling and resource management. Compiler vendors may have less experience in these areas and they have not been effective in merging the experience bases of compiler writing with executive writing. It is therefore important to evaluate the overall performance of the execution-time characteristics of Ada programs. The remainder of this chapter gives some of the characteristics that must be considered in an evaluation.

Figure 6-1: Three Models of Ada Runtime Configuration

6.2. Time Efficiency of Generated Code

The efficiency of the code generated by the compiler may be investigated by inspection and testing. Inspection of code generated by simple programs provides anecdotal evidence of the quality and efficiency of the code. Much information can be gained from a few well chosen examples. Benchmarking, on the other hand, can be used in a "black box" fashion for more exhaustive and automated testing. Numerous optimizations are available to improve the efficiency of the generated code.

6.2.1. Inspection

Much can be learned by inspecting the assembly listing of the code generated by an Ada compilation system for some simple Ada programs. This strategy can be used to obtain approximate measures of the quality of the code generator and to determine the efficiency of various Ada features. It can also provide insights to the overall code generation strategy of the compiler. Required for code inspection are an assembly listing of the generated code, knowledge of the instruction set of the target computer, understanding of code generation techniques, and understanding of the requirements of the Ada language. This activity should be undertaken only by experienced technical people.

The following examples were presented at AdaJUG by Robert Firth [Firth 88] in December 1988. They were used to make some observations and recommendations about Ada code quality. The first example, shown in Figure 6-2, is designed to test the code for several operations on a simple record of three integer components. The operations are assignment, comparison, and aggregate assignment. By comparing the generated code with optimal assembler coding, observations can be made about the ability to take advantage of word alignments, optimal use of the instruction set architecture (ISA), use of unnecessary variables, use of unnecessary instructions, and unnecessary tests of impossible conditions.

```
type Triple is record
  x,y,z : integer;
end record;
                -- static aggregate assignment
tconst : constant Triple := Triple'(1,2,3);
t1,t2 : Triple;

  t1 := tconst;  -- record assignment
...
  if t1=t2 then ...  -- record comparison
...
  t1 := Triple'(t1.x,t1.y,t1.x);  -- record construction
```

Figure 6-2: Simple Test for Complex Types

A second example was the "if X in 1..10 then ..." code fragment. Firth observed that one compiler was generating a Boolean value unnecessarily, that there were numerous unnecessary instructions, and that the ISA was not used effectively. The underlying cause seemed to be a series of expansions of higher-level abstraction idioms into lower-level ones, with loss of efficiency at each stage.

A third example, shown in Figure 6-3, was used to observe the code generated by a simple procedure that copies a constant value into its parameter. This example can be used to observe conventions for parameter passing, stack handling, and register saving and restoring. In this particular example it was observed that the nineteen instructions on one particular ISA could be reduced to three by a number of optimizations.

```
procedure P (X : in out Integer) is
begin
  X := 1;
end P;
```

Figure 6-3: Simple Test of Procedure

Inspection of generated code should not generally be used as an exhaustive testing technique, but it is effective for making limited but useful comparisons of different compilers for the same ISA. Additional simple tests can be used for the code generated by exceptions, generics, declarations, and many other Ada features.

6.2.2. Testing

The second method of determining the efficiency of the generated code is to develop black box tests called *benchmarks*. This technique is less dependent upon the target machine than is inspecting assembly code and offers a systematic and semi-portable way of observing efficiency. However, it is subject to the caveats described in Chapter 8. Testing can be said to answer the "what" of the efficiency of the generated code, but does not answer the "why," as inspection does.

Each individual Ada language feature takes a certain amount of time to execute if it is treated in isolation. However, it is sometimes difficult to measure these times (because they are so short) and the time in isolation may not be relevant when the feature is used in combination with other language features. Also, the performance of each individual Ada feature may be a combination of the performance of the generated code and the performance of the runtime system. The more complex an Ada feature, the greater the chances that there will be some effect by the runtime system. Among the features that are *not* likely to generate calls to the runtime system are the following:

- arithmetic and logical expressions
- selection statements (**if** and **case**)
- loops
- subprogram calls
- selection from and assignment to composite data structures

Each of the above operations has thousands of alternative forms depending on the number of parameters, the size and complexity of the objects, and the levels of nesting. For example, it makes little sense to ask how long it takes to complete a subroutine call unless the number of parameters, their types, and their direction are also known. As a general exercise, the benchmarking of many forms of given Ada features quickly reaches a point of diminishing returns unless the user can characterize an application very precisely. It is better to get some general feeling for performance and perhaps use fine-grained tests for troubleshooting or special investigations.

Among the features that are likely to generate calls to the runtime system are:

- dynamic storage allocation and deallocation
- elaboration of data objects
- task creation and termination
- rendezvous
- delay
- input and output
- exception handling
- interrupts

These features can also be measured with the so-called language feature tests, and the number of parameters and combinations is just as great as those described above. The difference in this case is that the features are more opaque to the user. Whereas in the former case the evaluator could see precisely what is happening from an assembly language listing of the program, in the case of these Ada features, such an investigation will lead to a call to the runtime system. Unless the user has a source code license to the runtime system, the code that is being executed is subject to scrutiny only if the machine code is disassembled. Disassembled machine code is often difficult to interpret.

Fine-grained tests may also be useful for establishing whether the runtime system exhibits deterministic behavior. To accomplish this, however, the instrumentation must include high precision timing capability so that each instance of execution can be timed rather than averaged (see Sections 8.3 and 8.4).

6.2.3. Optimizations Supported

Many different techniques are available to compiler vendors for improving the efficiency of generated code. It is important for the evaluator to decide which of many potential optimizations are critical to the application and to test to ensure that these options are provided in the generated code. There are many opportunities for optimization in Ada, as there are in

all programming languages. The *RM* gives the rules in Sections 10.6 and 11.6. Other suggestions are provided by the *Ada Implementers' Guide* [Goodenough 86b] and the Ada *Rationale* [Ichbiah 86]. Since Ada is a complex language, a highly optimized compiler is very important for performance.

Among the optimizations that can be performed are the following:

- **Dead code elimination:** Code that is unreachable may be removed.

- **Folding:** Operations on operands whose values are known at compile time can be performed at compile time.

- **Common subexpression elimination:** Expressions that have previously been evaluated need not be reevaluated (can be performed within statements, within control structures, or globally).

- **Strength reduction:** Replacing complex, expensive operations with simple, less-expensive operations; for example, replacing multiplication with addition especially in the case of loop indices used in array subscripts.

- **Expression simplification:** Application of valid mathematical or logical laws such as associativity, commutativity, multiplication by 1, etc. to simplify arithmetic and logical operations.

- **Cross-jumping:** Merging of common code sequences at the end of conditional branches.

- **Code motion:** Moving common code sequences so that they are not repeated in conditional branches and loops and so that they do not cause unnecessary jumps.

- **Peephole:** Reduction of short machine code sequences by passing a "window" over the final object code to eliminate or collapse adjacent instructions, and to substitute more efficient instructions wherever possible.

- **Habermann-Nassi transformation:** Technique for reducing the number of context switches required to execute a rendezvous.

- **In-lining:** Short subprograms may be placed in-line even without a pragma.

- **Static elaboration:** Certain objects whose characteristics are known at compile time can be elaborated before execution time.

- **Global optimizations:** Because of information in the Ada program library, optimizations can be performed across compilation units.

These are by no means all the optimization techniques that can be performed and are listed here only to provide a representative list. The ACEC test suite described in Section 9.2 has the most comprehensive set of tests for evaluation of optimization techniques. The *ACEC Reader's Guide* [Wright Research and Development Center 88c] lists over 20 optimizations that are tested for in the ACEC test suite.

It is not always clear to an evaluator what, if any, optimizations are important to the evalua-

tion process. Optimization tests are very fine-grained and may provide little useful information that is not provided by a coarse-grained test. Unless the user has some idea of a model for what optimizations could be performed on typical application code, the information on what optimizations are performed may be useless. For example, the user may wish to determine the optimizations that can be performed on a **case** statement. On the other hand, the number of optimizations performed may provide a good indication of what the general level of optimization is. Furthermore, certain optimizations such as Habermann-Nassi and automatic in-lining may be isolated as particularly important to an application that makes heavy use of tasking and small subprograms respectively.

Some implementations provide pragmas that facilitate optimization. The user specifies by a pragma that certain coding restrictions will be adhered to. This allows the compiler to generate code that is less general than if no restrictions were imposed. Examples are pragmas for fast interrupts or for certain kinds of tasking paradigms.

6.3. Space Efficiency of Generated Code

> *The size of the generated code is usually measured in terms of bytes per Ada statement. Since this measure varies greatly from statement to statement, it is important to have a uniform and representative program for which this measure is applied. One should be very wary of "code expansion" ratios unless one knows what is being measured.*

As was described in Section 6.1, the amount of memory required by a Ada program depends on two things: the object code generated by the compiler and the runtime system. Each of these is discussed in turn. While it may seem that the runtime space required is a constant (and often is in practice), it need not be, and the ability to configure the runtime system is one of the most important requirements of space-limited applications.

From the point of view of an evaluator, the space requirements are probably best measured with a representative application benchmark. In that way, the evaluator can use the code expansion metric with a fixed program over a set of compilation systems. Code size in units of STORAGE_UNITs can be measured in a portable way using the difference between the ’ADDRESS attribute of two labels. One problem with this solution is that not all implementations support the ’ADDRESS attribute. A machine-dependent solution is to write a simple assembly language function that returns the address of its caller. Both these methods are provided to compute code expansion sizes in the ACEC. Another problem with this technique is that code motion optimization may move some code. This is not a serious problem at present since few implementations support code motion, but it may prove to be more troublesome as optimizations become more sophisticated. Finally, the evaluator must be cognizant of the effects of elaborated data structures and elaboration code.

6.4. Time Efficiency of the Runtime System

The runtime system is code which is not generated from the application program. Rather, it is code that is needed by the generated code to perform functions such as task management, memory management, exception management, and input/output. The efficiency of the runtime system can be more important than the efficiency of the generated code, because the user may have less control over it.

6.4.1. Tasking

Tasking is the mechanism for concurrency in the Ada language. The performance of the implementation of tasking will be of importance for those applications that choose to use tasking and of little importance to those applications that choose not to use tasking. Among the features critical to performance are:

- task creation and termination
- simple rendezvous (task synchronization)
- selective waits, conditional and timed entry calls
- delay statements
- abort statements
- priorities

Rendezvous can be accomplished with varying parameters, varying calling and accepting alternatives, varying states of guards on selects, and varying synchronization conditions (the order in which tasks get to their synchronization points).

Of particular importance to an evaluation of the tasking implementation is the performance under load. How does performance degrade as more tasks become active in the system? How does performance degrade when there are more tasks waiting on an entry? How does performance degrade when there are more alternatives in a selective wait? How does performance degrade with increased number and size of rendezvous parameters?

Other questions have to do with implementation dependencies. How many priority levels are supported? How are alternatives selected from among the open alternatives? What scheduling algorithm is used for tasks? How does the delay respond to a parameter that is close to zero? What optimizations are performed with respect to tasking? The evaluator should determine which questions are relevant for the application being developed and then conduct tests to answer those questions.

6.4.2. Exception Handling

Exceptions are meant to be used for rare events. As such, the overhead for using exceptions should be low when the exceptions are not raised. Thus, the runtime system overhead of entering and leaving a frame (block, subprogram, package, task unit, or generic unit) in which an exception is defined should be very small.

The overhead of the following operations can be measured for both user-defined and predefined exceptions:

- declaring the exception
- raising the exception
- handling the exception
- propagating the exception

These operations are normally handled by the runtime system. They have many forms and alternatives. Applications making heavy use of exceptions should carefully evaluate the time and space of the exception mechanism.

6.4.3. Input and Output

I/O is provided in the language by means of predefined packages or vendor-supplied packages. The predefined packages are SEQUENTIAL_IO and DIRECT_IO (generic packages) for I/O operations on files containing elements of a given type, TEXT_IO for text I/O, and LOW_LEVEL_IO for direct control of peripheral devices. Package IO_EXCEPTIONS defines the exceptions needed by the I/O packages. Rarely will an application use all these packages. For example, embedded applications may use only LOW_LEVEL_IO. However, there are runtime overheads associated with all I/O operations and these should be evaluated for the relevant packages. I/O performance is particularly important to MIS applications.

Ada provides a FORM string parameter to CREATE and OPEN procedures that permits users to specify values of such timing parameters in an implementation-dependent fashion. The speed of programs using default values may not be optimal. The evaluator must be concerned with setting these parameters in test programs to "reasonable" or "comparable" values.

It should be noted that packages provided in the language standard were not expected to yield high performance implementations. The intention was more to provide uniform ways of performing a minimal set of I/O operations. As a result, I/O is implementation dependent and there is considerable variation in I/O functionality and performance. For example, I/O for unconstrained arrays and variant records is not always available. Parallelism of I/O in a multitasking situation is implementation dependent. Finally, it should be noted that I/O benchmarks may be influenced more by the timing parameters of I/O hardware and operating systems than by runtime software. Operating system effects include multiple buffers,

read-after-write checking, read-ahead, shared file access, disk allocation schemes, block sizes, etc.

6.4.4. Elaboration

Elaboration is defined in *RM* 3.1(8) as the "process by which a declaration achieves its effect." The time and space required to execute declarative items such as type declarations and simple or complex object declarations can be measured with the same techniques used for other language features as described above. The *RM* also states in the same paragraph that elaboration "happens during program execution." However, it is well recognized that in many instances, all the information that is needed to elaborate an Ada object is known at compile time. Then valuable execution time can be saved if those declarations that can be elaborated at execution time are in fact "pre-elaborated." For example, arrays can be pre-elaborated and initialized if the array dimensions and initial values are known at compile time. This is a common, but by no means universal, optimization that can be performed at compile time.

Library units must be elaborated in an order consistent with the partial ordering defined by the unit dependencies. Pragma ELABORATE can be used to provide some user control over the elaboration order defined by the partial ordering. Since the elaboration of a library unit only happens once during program execution, it is not as easy to measure the time required for elaboration of a library unit. This is because the operation cannot be placed in a loop so the "dual loop" paradigm cannot be used for reliable timing measurements. Library unit elaboration time may be highly variable from compiler to compiler and also from elaboration to elaboration. If declarative items in the library unit have not been pre-elaborated, the time on a host-based system may depend on the file system or the paging system or both. The time required to allocate dynamic objects within library units is similar to that required by explicitly invoking the "new" allocator. This can be time-consuming and varies considerably between implementations.

The importance of elaboration to an evaluation is highly variable and depends on both the application and programming style. One important consideration is how much of the elaboration is done at compile time. For certain applications this can be an optimization that overshadows some of the other optimizations because it is so pervasive. For real-time applications that need to restart often (forcing re-elaboration) or change modes by loading a new program, the elaboration time may be critical and deserve careful attention. For non-real-time applications, the elaboration time may be inconsequential. A heavily nested programming style will force more elaboration at different points of program execution than a flat program style where most of the elaboration activity will happen at the beginning of execution. This means that the overhead of restarting a program may depend on the efficiency of the implementation, the elaboration options provided, and the style of programming. If elaboration is an important consideration, it should be systematically tested using hardware monitors so that the design tradeoffs are understood.

6.5. Space Efficiency of the Runtime System

> *Memory space is required for the runtime system code (determined at load time), for implicitly created entities, and for explicitly created entities. The evaluator must be concerned with the amount of space required, the method of allocation and deallocation, and the recovery of unused space.*

Executing Ada programs require storage for code and data. The major areas of concern for an evaluator are:

- static space for application code and data
- runtime system space
- space for implicitly created dynamic entities
- heap space for explicitly created dynamic objects

Static space for application code and data is covered in Section 6.3. The other three categories are controlled by the runtime system. The space taken by the runtime system itself is highly variable from compilation system to compilation system.

Storage requirements are especially important in embedded systems that do not have virtual memory capability because the space taken by the runtime system is not available for application code. A large runtime system may affect the program size to a much greater degree than the size of the generated code. A full runtime system may require over 100K bytes of storage, but a minimal one for a minimal function application program may require only a kernel of 1K or 2K bytes. For space-critical applications, it is important to be able to configure the runtime system either automatically or by hand. The straightforward method of constructing a loadable program is to include the entire runtime system. If large parts of the runtime system are unused, then this is extremely wasteful of space. Thus, it is necessary to be able to include those portions of the runtime system that are used and to exclude those parts that are not used. This is accomplished by a vendor-supplied "selective loader," as described in Section 7.2.

There are no automated or semi-automated ways to determine the size of the runtime system. This information should be available from the load map produced by the linker or in the vendor-supplied documentation.

Space is implicitly required for the following types of Ada operations:

- creation and initialization of tasks
- entry into a new scope
- handling of an exception

The runtime system must allocate and deallocate the space for these Ada operations as well

as for dynamic structures using allocators. The language does not define strict requirements for allocation and deallocation of space. Two possible hazards are the failure to reclaim space and fragmentation of space. Programs compiled by some first generation compilers were known to run out of space because certain Ada features allocated space which was never returned to the free list. Since the ACVC does not systematically test for performance or capacity, the test programs all ran to completion because they did not exceed the storage limits, but real programs failed simply because they executed a simple Ada feature multiple times. These problems can be uncovered either by running systematic tests or by running large programs that continuously exercise a substantial subset of the language. Some tests in the suites described in Chapter 9 do contain runtime capacity tests.

Ada also permits objects to be explicitly created with an allocator and explicitly designated as free with the predefined, generic, library procedure UNCHECKED_DEALLOCATION (if supported by the implementation). In addition to determining the efficiency of these operations for different kinds of objects, an evaluator should determine whether and under what circumstances the implementation performs "garbage collection" (the recovery of unused storage space). *RM* 4.8(7) states that "an implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible." If there is no garbage collection, the user must restrict the amount of dynamic storage allocated or provide a user-defined package for dynamic storage management. If garbage collection is performed, it can be done dynamically (as each object is deallocated), periodically (after a certain amount of time), or when the space pool is exhausted. Real-time applications can rarely tolerate garbage collection at unpredictable times so the evaluator should determine the impact of this runtime feature.

6.6. Features of the Runtime System

> *In addition to being fast and small, the Ada runtime system may have a number of features that are required by certain application domains to meet functionality or performance constraints.*

There is an inevitable tradeoff between generality and performance. If the application domain does not have severe performance constraints, then a general purpose runtime system can serve the needs of a variety of applications. For embedded and real-time applications it may be necessary to sacrifice generality for additional performance. Some of the functionality and performance features that may be useful for such applications are listed below.

- **Configurability:** It is *not* the case that one runtime system is capable of efficiently supporting all application domains. Thus, to achieve maximum performance it may be desirable to allow the user the flexibility of modifying the character of the runtime system to suit application requirements. This is particularly important when application requirements change in the course of a project. Among the characteristics that the user may wish to change are the timer characteristics (SYSTEM.TICK), scheduling options, heap and stack sizes, maximum number of tasks (if static), etc.

- **Alternative tasking/synchronization support:** It has been proposed, but by no means universally accepted, that alternatives to the Ada tasking be provided for applications having the most stringent timing constraints. Real-time executives have provided such operations for many years and it is a more familiar paradigm for many real-time programmers. Operations include task creation, deletion, suspension, and resumption as well as synchronization primitives implemented as semaphores, mailboxes, or events.

- **Distributed system support:** There are a number of Ada implementations that currently are supported on closely coupled (shared memory) distributed systems. On the other hand, there are few, if any, runtime systems that provide direct support for Ada programs running on a loosely coupled (non-shared memory) systems. Runtime systems can provide support for federated systems in which separate Ada programs run on each node of a distributed system or for a unified system in which a single Ada program runs on the entire network.

- **User "hooks"**: Some runtime systems provide a user the capability of gaining control when certain operations are performed. This capability allows runtime enhancements, performance monitoring, debugging, etc. For example, user hooks could be provided for the task creation, task switching, or task deletion runtime operations. This would give the user an opportunity to perform memory management processing or to time the overhead of a task switch.

- **I/O support:** Asynchronous I/O to physical devices must be supported for many real-time applications. The runtime system should provide I/O support that presents a device-independent interface to the user. The interface should be configurable with respect to device types and provide the necessary mechanism for applications to be notified of asynchronous I/O operation completion and status.

- **Co-processor support:** The runtime system should make it possible to support special purpose co-processors for enhanced performance. Examples include floating point co-processors and memory management units. Either the compilation system should support these co-processors directly or provide the necessary "hooks" for the user to implement this support.

These, and many other features, are covered in much more detail in reports produced by the ARTEWG [Ada Runtime Environment Working Group 87, Ada Runtime Environment Working Group 88a, Ada Runtime Environment Working Group 88b].

6.7. Implementation Dependencies

Many options exist for implementing Ada runtime functionality. These are documented in ARTEWG's Catalog of Runtime Implementation Dependencies (CRID) [Ada Runtime Environment Working Group 87]. The evaluator should consult the CRID and determine whether any of the implementation dependencies are critical to the application.

There are many places in the language definition where Ada implementors are free to choose how to implement a language feature (as long as the feature conforms to the rules of the language). Some of these choices (such as the evaluation of operands in an expression) may have little effect on the overall time/space tradeoffs. Other choices (such as sharing of code bodies for generics or automatic in-lining of procedures) may have a significant effect on overall time/space tradeoffs. It is important that an evaluator know what performance impacts the choices will have on the application.

Another source of information on implementation dependencies is the *ACVC Implementors' Guide* [Goodenough 86b]. This report was produced as part of the Ada Compiler Validation Capability and it explains the consequences of the language rules. It was meant to provide guidance for writing test programs, but has also served to provide useful information to designers of implementations. Both the *Implementors' Guide* and ARTEWG's original *CRID* are organized by *RM* section, making it easy to find implementation dependencies of interest. A newer version of the *CRID* is organized by functional area and has a very large index so issues can be located by AI number, *RM* section, pragma name, etc. These reports, along with the *RM*, can serve as handy references for evaluators looking for insight into performance anomalies.

Some of the important runtime implementation-dependent issues being considered by the URG are the following:

- garbage collection policies
- scheduling policies, including time-slicing and non-blocking I/O

- guidelines for DURATION and delay (0.0)

- guidelines for elaboration ordering

As is pointed out in Section 5.5, these types of issues are significant for evaluation if the application has specific requirements in these areas or when portability is a primary concern.

6.8. Interrupt Handling

> *Interrupt handling is heavily dependent upon the implementation. There are many implementations and many options for optimization. The evaluator should understand how interrupts are handled in a particular implementation and determine whether this is sufficient for the application.*

Handling interrupts in Ada is always machine dependent. Depending on the machine architecture, the compilation system, and the system designer, there are numerous alternatives. Most of the alternative solutions are not portable between target machines or between different compilers for the same target. The language standard approach to interrupts is to use task entries and address clauses. In this approach an address clause is used to associate the interrupt address with the interrupt entry. Some systems do not provide this functionality and instead use machine code insertions or a system call to load an interrupt table with the address of an Ada procedure. These options and their variations are described by Doug Bryan [Bryan 88].

When applicable to the system being developed, the functionality and performance of interrupts provided by the compilation system should be carefully evaluated. Because interrupt handling is so machine-dependent, there is no portable code to test interrupt handling; hence, it is largely ignored in the benchmark test suites described in Chapter 9. Among the questions to be asked are the following:

- What language support is there for handling interrupts?

- Are there language pragmas for "fast interrupts" (giving the user the opportunity to give up some tasking functions for improved performance)?

- How much time is required to get to the interrupt service routine from the time of the interrupt?

- How much time is required to return to the interrupted program from the exit of the interrupt service routine?

- Are the times above deterministic (i.e., does the operation take the same amount of time each time it is executed)?

- How are nested interrupts handled?

- How do interrupt priorities relate to hardware priorities and Ada task priorities?

- Are there limits to what one can do in an interrupt service routine with respect to accessing data and synchronizing with or activating other tasks?

- Is there sufficient support for representation specifications (Chapter 13 features) to allow access to hardware for interrupt service?

- How long are interrupts disabled by the runtime or generated code in the worst case for each specific operation?

6.9. The Clock and Timing Issues

> *For real-time embedded systems, the abstraction of time is of critical importance. The evaluator must determine how time is represented, what the resolution of the clock is, and what the costs of invoking clock services are.*

Ada's abstraction for time is contained in a predefined package CALENDAR. This package contains an implementation-dependent type called TIME. The package compares times and performs arithmetic operations on times which are clearly meant to be of coarse granularity. The constructors and selectors for time break time into years, months, days, and seconds (and fractions of seconds). The function CLOCK returns the time of day. Package SYSTEM defines an implementation-dependent type called DURATION. DURATION is a fixed point type whose values are expressed in seconds. The type must allow representations of durations (positive and negative) of up to 86400 (number of seconds in a day) and the smallest representable duration must not be greater than twenty milliseconds. *RM* 9.6(4) recommends that the value should not be greater than 50 microseconds "whenever possible." Since it takes 18 bits to represent the integer part and 14 bits to represent durations down to 61 microseconds, it can be assumed that the recommendation was based on the abilities of 32-bit architectures.

For many applications, the abstraction of time is not of critical importance. For printing the time of day on a user console or measuring the time required to execute functions that take seconds or more, package CALENDAR is quite sufficient. For real-time applications that measure time in microseconds and milliseconds, the abstraction for time becomes extremely important.

For real-time applications the evaluator should determine the following characteristics of the Ada implementation:

- The implementation-dependent representation for type DURATION.

- The implementation-dependent representation for type TIME.

- The speed at which the clock ticks. (This should be the same as SYSTEM.TICK, but has been found to be otherwise in some implementations.)

- The maximum and minimum possible times between the expiration of a delay and the rescheduling of the task executing the delay. (Note that this rescheduling requirement says nothing about when the task will restart execution, since its priority may be such that other higher priority tasks are eligible for execution at the time the delay expires.)

- Any additional packages provided by the vendor to provide a more precise definition of time.

- The relationship between the Ada clock and the system (hardware) clock.

The application developer may be unable to use the Ada facilities for timing. In that case, there is the option to write an interface to access a hardware clock that is provided with most embedded computers. A hardware device can normally be programmed so that the Ada interface can set the clock, interrogate the clock, start the clock to count down, and interrupt when it reaches zero. Such clocks are often accurate to one microsecond or better. An example of how Ada can be used to interface with a hardware clock is given in [Borger 87].

7. Support Tool Issues

> *An Ada compilation system includes more than just a compiler. There is a minimal tool set without which the compiler becomes almost useless. This tool set is often called a "minimal Ada programming support environment" or MAPSE. Support tools should provide adequate functionality and performance, have a good user interface, and be easy to use with one another.*

This chapter will attempt to show the importance of evaluating four important support tools, the program library system, the linker/loader, the debugger, and the target simulator. The first three are normally considered to be part of a minimal tool set (MAPSE) and are essential for producing correct programs on the target. The fourth is considered to be an important tool for cross-development systems. While these three tools represent only a small part of a complete Ada Programming Support Environment (APSE), they are among the most important tools and the ones most tightly coupled to the compiler itself. For a more complete discussion of APSEs, the reader is referred to the original Stoneman requirement [United States Department of Defense 80] and a description of one of the early government funded environments [Wolf 81]. For specific detailed requirements and criteria for evaluating these support tools, the reader is referred to the SEI environment evaluation work [Weiderman 87a] and the *Ada Evaluation System* [Marshall 87]. Future versions of the ACEC will cover these areas as well.

7.1. Program Library System

> *Many of the functions of the Ada program library system are given by the language definition. These functions support the separate compilation of program units, which facilitates "safe" top-down and bottom-up programming. Because Ada compilers must do a great deal of checking across compilation units, the structure and performance of the library system is critical to compile-time performance.*

Ada programs may be broken down into separate modules to facilitate large development efforts. However, unlike some other languages that permit independent compilation, each Ada compilation depends on information in a program library so that knowledge about the properties defined in other modules is available to the module currently being compiled. For example, the number and type of parameters of a subprogram call can be checked with the definition of the subprogram that was previously compiled into the library. For a better understanding of the purposes and structure of the Ada library system, the reader is referred to the Ada *Rationale* [Ichbiah 86].

The purpose of evaluating program library systems is to determine how the library mechanisms can affect programmer productivity. While the implicit library usage by the compilation system is part of compile time evaluation (see Chapter 5), this section will try to elucidate the evaluation issues with respect to explicit library usage.

The following facilities are not defined by the language, but are expected to be present in the programming support environment:

- **Library creation and deletion:** Although a library is normally created from scratch (empty), facilities that create it with a program or family of programs from another library are desirable. How long does creation take? How much space is consumed by an empty library? Can a library be deleted without first deleting all its contents?

- **Inclusion of library units:** There should be a command to include a unit of one library in another library. Can this be done without recompiling the unit?

- **Deletion of library units:** There should be a command to delete a unit from a given library. Is the space made immediately available for reuse? Does deletion or updating of a unit "obsolete" executable files that use it?

- **Completion check:** There should be commands to check whether some units of a program are obsolete or missing. Can valid compilation orders be provided? Can the system (re)compile all the units of a program that would be required by a set of source changes without the user's having to explicitly identify the impacted units? (i.e., perform the function of the UNIX command "make")?

- **Status commands:** There should be commands that allow a user to display global information about the current state of the library, such as the units in the library and their dependency relationships, whether the units have been compiled, and which units need to be recompiled. If space is allocated to a library, can the amount of available space be listed? How much space is allocated to each unit? When was a unit last modified?

- **Library structure:** Does the librarian require the source to be in the same directory as the library files? How transparent is the library with respect to the file system of the operating system? Is concurrent access by multiple users permitted?

7.1.1. Recompilation and Incremental Compilation Features

Whenever any unit in the program library is compiled, it *may* invalidate those units that depend on it. For example, if a subprogram specification is changed, all those units that use that definition are subject to recompilation. However, it may not be not necessary to recompile all the units that depend on the library unit, if they do not depend on the specific definition that was changed in the library unit. For example, if unit X only uses subprogram A from a unit that exports subprograms A and B, then recompilation of X is trivial when only the spec of B is changed in that unit. Thus, it is extremely important to determine the granularity with which recompilation is required. The simpleminded and straightforward solution is to simply mark a compilation unit as needing recompilation without regard to the recompilation impact. A more sophisticated approach is required to determine the smallest fragment of the program that requires recompilation.

Incremental compilation is the ability to use information about previous compiles to perform new compiles at decreased cost. If an environment can recognize that the only change to a source file is a comment, recompilation is again a trivial operation. If an environment can recognize that the only change to a source file is the addition or deletion of a definition in a package specification, then generally none of the other definitions should need to be recompiled. Changes to bodies of units should not force recompilations of other bodies or specifications. It should be noted that such avoidance of compilation and incremental compilation are not generally provided in today's compilers. Good intermediate representations such as DIANA are needed to capture and reuse work that has been done by a compiler.

7.1.2. Sublibraries

Another feature that is desirable to promote programming-in-the-large with multiple groups is the concept of shared libraries for different subsystems. This capability should permit users to link two or more libraries so that when a compilation is started in one library, the compiler can be steered to another library to find parent units. This feature does raise the consistency problem in that when libraries are linked in this fashion, the recompilation flags must be propagated across library boundaries when a unit is changed in one library and is depended upon by a unit in another library.

7.2. Linker/Loader Support

> *The linker/loader support tools provide the capability of combining separately compiled Ada units into a single module and preparing for execution by loading the module into memory. This process should be efficient in both time and space.*

The linker/loader support tools may be provided by the underlying operating system on the host or by the Ada compilation system. When they are provided by the underlying operating system, they have the features and performance provided by the operating system vendor. If they are provided with the compilation system, they may provide additional features and performance. Some environments are so tightly integrated that the linking process is essentially invisible because the programs are linked incrementally. With a host-based development system the linker and loader may be tightly coupled, but in a cross-development system the loader is necessarily a separate and highly machine-dependent step involving programs running on two machines.

7.2.1. Selective Loading

Ada permits developers to use object oriented techniques whereby similar objects and operations are combined in a "package." For example, a package could consist of a library of trigonometric functions or a set of graphics objects and operations. When a program unit **with**s one of these packages, it is not necessarily the case that all the operations supplied with the package are used. Therefore, it would be more efficient if the linker/loader included in the load module only those subprograms that are actually referenced by the object program. As an example, for very large packages such as a vendor supplied mathematics package, the difference in the runtime code could be the difference between a 100-byte

routine for returning an absolute value and a 10,000-byte package for computing most standard mathematical functions. A similar consideration applies to the selective loading of only those modules in the runtime system that are referenced by the object program. The absence of selective linking features tends to create load modules of hundreds of thousands of bytes when only tens of thousands of bytes may really be required.

7.2.2. Other Linker/Loader Features and Options

Other features that may be supported by the linker/loader are the following:

- **Memory assignment:** There may be a means of specifying the placement of the code and/or data of the program in particular memory locations so that assignments to read-only memory (ROM) may be accommodated or so that code segments or entry points can be aligned in memory.

- **Partial linking:** There may be a means of partial (incremental) linking for programs so that small changes at the highest level do not require relinking the entire program.

- **Dynamic loading/overlays:** When the load module is too big for the available memory, are there any automated capabilities to load portions of the program as they are needed?

- **Link-time optimization:** Most optimizations take place at compile time, but the separate compilation capability prevents the compiler from having complete information about the program. Additional optimizations are possible at link time. For example, additional dead code can be eliminated.

- **Linking to other languages:** Support for interfacing with subprograms and objects in other languages must be provided by the compiler using pragma INTERFACE or by other import/export techniques supported by the implementation.

- **Dynamic memory allocation:** Is it possible to specify how much memory is to be allocated to various dynamic structures such as the stack and heap?

- **Library searching:** Does the linker support library searching to satisfy external references not resolved in the primary library?

In addition to these features, the following generic concerns should be considered in the evaluation of a linker:

- performance

- capacity

- informational outputs, including the link map

- diagnostic outputs

- user interface

7.2.3. Downloading for Cross-Development Systems

For cross-development systems, the loading of an Ada program onto the target requires two programs, a downloader running on the host system, and a receiver program running on the target system. The cross-development package must also support triggering (starting) of the execution of the application code on the target. The downloading process is often time-consuming, complex, and error prone. The performance of this communications link can have a significant impact on the ability to evaluate a compilation system (by running benchmark programs) and later to develop systems. In fact, the time taken to download a program may, in some cases, be longer than the compile, link, and execute times combined. Download time may also be increased by as much as a factor of two when hardware/software monitors or microprocessor development systems are used because of the need to download additional symbol table information and the formatting information associated with it. Download time may be decreased if static parts of the loadable image such as the runtime system do not have to be reloaded for each program execution. This critical download link depends primarily on the speed of the hardware connection, but must be considered carefully by cross developers. What is important to determine is whether the vendor supports the exact target/link configuration to be used. If not, the user can expect to have to customize the downloader and receiver to the particular configuration being used.

7.3. Support for Debugging

> *Debugging should take place at the highest abstract level possible, which means that a debugger should be integrated with the compiler to provide source code information to the user. Because debuggers may improve programmer productivity significantly, it is important to have at least a minimum level of debugger functionality.*

It is useful to distinguish three levels of debuggers. A machine-level debugger knows only about machine addresses, machine instructions, and contents of machine locations. It is useful to an Ada programmer only with the memory maps generated by the compiler and linker and only for testing the generated code in assembly language format. A symbolic debugger is essentially a machine-level debugger with symbolic information available so that the programmer can refer to data objects by name rather than by machine location. A source-level debugger allows the programmer to display source code and to enter all debugger instructions in terms of source code instructions. The source-level debugger provides a high level of abstraction to maximize programmer productivity.

There are no benchmark tests for debuggers. Since debugging is a highly interactive activity, there are two approaches that are used to evaluate the functionality and performance of a debugging system. The first is a checklist of features and performance characteristics and the second is a debugging "scenario" that permits the user to take a program and a set of tasks to be performed and conduct a debugging session. The checklist is often filled out using only the documentation, while the scenario must be conducted on the system being

evaluated. Both of these techniques are somewhat subjective, but the second is preferred because it promotes exploration of the system and allows serendipity and discovery not facilitated by the "hands-off" approach.

Among the operations important in debugging are the following:

- Examining the control flow (break operations)
    - after n statements executed
    - at a particular statement
    - at program unit entry or exit
    - at the raising of an exception
    - at a scheduling point
    - on modification of a variable (tracepoints)
    - on interrupt by the programmer
    - after the iteration of a loop
- Examining the program state
    - data objects
    - call history
    - active tasks
    - status of blocked tasks
- Changing the program state
    - modify data object
    - add, modify, delete code
    - restart at designated instruction
    - selectively step into or skip entire subprogram
    - raise an exception
- Displaying debugging information

- displaying source code
- displaying assembly code
- displaying breakpoints
- displaying tracepoints

Debugging in a multitasking environment poses special problems and may require control of time (because of conditional waits, for example). More complete checklists and example scenarios can be found in [Weiderman 87a]. As with most tools, a poor quality debugger can hinder rather than facilitate progress.

7.3.1. Effects of Optimization on Debugging

Optimizers tend to confuse debuggers. The reason for this is that the optimizer may change the order of operations in a program as long as it does not affect the semantics of the program. Thus, the source level model of the semantics does not map onto the runtime semantics. For example, an optimizer may move an invariant assignment statement outside a loop. If the user sets a breakpoint on that assignment statement, it may be difficult or impossible for the debugger to stop each time through the loop. Similarly, the values of variables may be different from what may be expected by looking at the source code because the object code may hold temporary values in machine registers rather than storing them into main memory. These problems may be solved in several ways:

- Disallow debugging for optimized programs.
- Limit the functionality of the debugger for optimized programs.
- Permit the debugger to have unpredictable behavior for optimized programs.
- Take extraordinary steps to ensure that the debugger works identically for optimized programs.

None of these options comes without limitations and costs. It is important for the evaluator to know which of the options the vendor has chosen, to understand the tradeoffs, and to know the implications.

7.3.2. Debugger User Interface

Just as the user interface is an important part of the compiler, the user interface is also an important part of the debugger. The debugger should be easy to use and easy to learn. It should have documentation and online help facilities. It should have clear informational and diagnostic messages. Links to the editor are also helpful.

7.3.3. Cross Debuggers

In a cross-development environment the debugger should execute on the host with a debug kernel on the target. The debug requests and the debug output should be sent back to the host where the user interacts with the debugger. The advantage of this is that the host with its considerable resources can be used to do most of the work instead of the limited resource target. The evaluator must determine the time and space implications on the target system of such a cross debugger. Other cross debuggers operate without a kernel on the target system. Instead they use hardware to monitor the target system bus. This form of debugger is the least intrusive and allows debugging of exactly the code that will exist in the production system.

7.4. Target Simulator

> *Target simulators can increase programmer productivity when operating in a cross-development environment.*

Cross-development systems are more complex than host-based systems. Programmer productivity can be lowered because of the time required to download programs and because of the vagaries of the hardware and the special hardware interfaces. A target simulator allows the user to execute programs for the target on the host system and to shorten the turnaround time for each run. While target simulators may execute at greatly reduced instruction speeds, the turnaround time between consecutive runs may compensate for this. It is important that a target simulator provide an almost exact duplicate of the environment provided on the actual target. In particular, the simulator must simulate time precisely both with respect to instruction speeds and clocks, it should have a debugging capability with exactly the same features and user interface as the cross debugger, and it must have the ability to simulate interrupts as they would occur on the real target.

The target simulator should do the following [Weiderman 87b]:

- Accurately simulate both the functional and temporal behavior of the target's instruction set architecture.
- Provide access to all memory locations and registers.
- Support typical features of symbolic debuggers.
- Perform timing analysis.
- Support simulated input/output interaction.
- Facilitate setup and reuse of test sessions.

Users should have a high degree of confidence in the quality and correctness of the tool.

7.5. Other APSE Tools

There are a number of other APSE tools that could be considered useful and deserving of evaluation along with the Ada compilation system. A primary consideration, if present, is how well they are integrated with the compilation system.

In addition to the compiler, debugger, and linker/loader, the Stoneman requirement [United States Department of Defense 80] for a Minimal Ada Programming Support Environment (MAPSE) calls for the following tools: text editor, pretty printer, set-use static analyzer, control flow static analyzer, dynamic analysis tool, terminal interface routines, file administrator, command interpreter, and configuration manager. Since this early attempt at defining requirements of an APSE, it has become clear that some of this functionality is more appropriately provided as an option of another tool (pretty printers and set-use analyzer functionality can be provided by compiler options) or incorporated into a more powerful tool (static control flow can be done with a browser and dynamic analysis can be done with a debugger). Other tools may be important at the requirements definition and testing phases of the life cycle. These include document processing systems, spelling checkers, project planning aids, presentation graphics tools, cost tracking and accounting systems, archival storage management systems, testing tools, electronic mail systems, performance monitoring tools, accounting systems, etc.

What Stoneman does make clear is that a tool set should make it easy for a programmer to move easily from one tool to another. This may be accomplished by a common representation for Ada programs such as the Descriptive Intermediate Attribute Notation for Ada (DIANA), which stores all the syntactic and semantic information about a program unit.

It is beyond the scope of this handbook to provide guidelines for the evaluation of other tools that should be tightly integrated to the Ada compilation system. The existence of a comprehensive integrated tool set may be part of the criteria for compiler selection. It certainly can have an impact on programmer productivity. Another desirable criterion for an environment is *openness*, the ability to add home-built tools or tools from other vendors. This criterion is often in conflict with tight integration. For further information on general APSE evaluation criteria, the reader is referred to the environment evaluation literature [Feiler 88, Lyons 86, Weiderman 87a, Wright Research and Development Center 88a, Wright Research and Development Center 88b].

8. Benchmarking Issues

> *Benchmarking is a black art. Benchmark design and development, as well as the use of benchmark data, require careful and painstaking analysis by skilled technical people. Simple acceptance of raw comparisons without an understanding of the tests and the testing environment is risky.*

Benchmarking is perhaps the most widely used performance evaluation technique. It consists of running a set of programs on a system to compare its performance with other systems. In the case of Ada, the purpose is to compare one Ada compilation system with another Ada compilation system or possibly with the compilation system of some other language. A problem that benchmark users should be aware of is that it is difficult or impossible to isolate the compilation system from the other components of its environment, namely the computing environment. This computing environment includes both hardware and software that may be difficult to control. As is pointed out by Dongarra [Dongarra 87], "bad benchmarking can be worse than no benchmarking at all."

Using benchmarks is like using statistics. If applied properly, they can enlighten. If used improperly, they can confuse, obfuscate, and deceive. Shepherd and Thompson have paraphrased the famous quotation attributed to Disraeli by Mark Twain ("There are three kinds of lies: lies, damned lies, and statistics") in a technical memo entitled "Lies, Damned Lies, and Benchmarks" [Shepherd 88]. Other technical papers providing insight into the potential and problems of benchmarking include [Clapp 86, Dongarra 87, Fleming 86]. The purpose of this chapter is to highlight what benchmark programs are and what they can and cannot do. It also provides some guidelines for those who may have to conduct benchmarking activities.

This chapter deals primarily with runtime benchmarks. The issues of benchmarking compile-time performance are covered more fully in Section 5.2.

8.1. Types of Tests

> *There are many kinds of benchmark tests. Users are not likely to have time to run all the tests that are available. Some understanding of the advantages and disadvantages is necessary in order to select those tests that provide the most useful information. Benchmarks can be small programs that measure an individual Ada feature (such as a subroutine call) or they can be large programs that measure many Ada features in combination. Fine-grained benchmarks are useful for pinpointing the strengths and weaknesses of an Ada compiler, while coarse-grained benchmarks are useful indicators of the overall efficiency as determined by the way in which individual features interact.*

8.1.1. Language Feature Tests

Language feature tests are meant to isolate a single or a small number of features of the Ada language. The idea is not to measure the time or space characteristics of the whole program, but rather to isolate a small portion of the program for measurement. Most, but not all, of the tests included in the test suites described in Chapter 9 are language feature tests. They attempt to measure how long it takes to execute a given Ada feature under various conditions. For example, there may be tests to determine the overhead of invoking a subroutine. But the time it takes to invoke a subroutine depends on the number and the nature of the parameters, so there may be dozens of tests that test many parameter combinations. One suite of tests has subroutine overhead tests for the number of parameters (1, 10, 100), the direction passed (in, out, in out), the type of parameter (integer, enumeration, string, various records and arrays of integers, and various sizes of parameters (1, 5, 10, 20, 100, 1000 storage units). Other tests exist for arithmetic, loop overhead, accessing components of record types, clock overhead, exception handling, task creation, and rendezvous. Just about every section of the *RM* has an associated performance test in one of the available test suites.

The advantage of language feature tests is that they may identify strengths or weaknesses in the ways that individual language features are implemented. If an application will make heavy use of a particular feature or set of features, then the user may wish to know which compilers perform best on that feature set. The user may be in a position to influence a vendor to make small changes in a compiler in areas where it is found to be deficient compared to other compilers. The disadvantage of language feature tests is that they inadequately address the impact of features being used in combination with one another. Just as correctness cannot be completely evaluated by a feature-by-feature test suite, neither can performance be completely evaluated by a feature-by-feature test suite.

8.1.2. Capacity and Degradation Tests

Related to language feature tests are tests of the capacity of a system (hardware and software) and tests to determine how a system reacts to increased loading. The former is determined largely by memory capacity and by internal limits. Examples are dynamic nesting levels and dynamic storage space. The latter has to do with how performance changes when heavy loads cause queues to increase or available memory to decrease. Examples are the effect on tasking operations of increased numbers of tasks and the effect on dynamic memory allocation of memory fragmentation. Exploration of these issues may require running a series of tests rather than a single benchmark.

8.1.3. Composite Benchmarks

Composite benchmarks are programs that are designed to test many features in combination with one another. Composite benchmarks may be small or large and may be application-dependent or application-independent. Generic tests such as the Sieve of Eratosthenes, Quicksort, Ackermann's function, or computing pi, are typical of academic programs written for numerical analysis or data structures courses. There is rarely any scientific basis for their selection as benchmark programs, but they tend to be widely available, which makes comparisons easy. Other tests tend to be larger and more application-

dependent. Examples include Kalman filtering applications, inertial navigation systems, radar tracking systems, or aircraft simulations. For real-time applications, buffer, relay, and monitor tasks serve as useful paradigms for composite benchmarks for tasking.

The advantage of composite benchmarks is that they *may* test a broad cross section of language capabilities (often they do not). They may also test just those features that will be utilized in the application to be written (often they do not). The primary disadvantage is that if the benchmark test produces a poor result relative to other systems, it is often difficult to determine the reason. It may be due to a single feature of the language, to uniform inefficiencies, or to the way the compilation system handles features used in combination. Fine-grained tests are needed to address these issues.

8.1.4. Synthetic Benchmarks

The general idea of a synthetic benchmark is to develop a skeleton application whose characteristics are typical in some way. Two of the best known synthetic benchmarks are the Whetstone [Curnow 76] and the Dhrystone [Weicker 84]. The Whetstone is constructed based on the static and dynamic instruction frequencies of 949 programs. It is meant to be typical of scientific numerical computation and is heavily weighted toward floating point operations. It is widely accepted as a means of comparing architectures, languages, and implementations of languages. The Dhrystone is intended to reflect the features of modern programming languages (e.g., record and pointer data types) and is intended to be typical of systems programs. It was originally written in Ada and is synthesized from static and dynamic frequencies of statements as determined by 16 different studies that analyzed large and small programs from a variety of sources.

Synthetic benchmarks can have many of the advantages of composite benchmarks. They test language features in a broad way, yet are in some sense more representative of a larger application domain because they have been constructed scientifically (i.e., based on many representative programs). The idea is to provide all the characteristics of many programs into a single program in exactly the proportions that they exist in some application. The disadvantages of synthetic benchmarks are that they are seldom accepted as being representative of any particular problem domain and they do not permit the user to isolate particular sources of inefficiency.

8.1.5. Application-Specific Tests

The most representative benchmark program for any particular application is the application itself. However, it is often difficult or impossible to benchmark a system with the real system because the software is not yet written. There may also be software dependencies and hardware dependencies that make it difficult to port the application to the system of interest. The next best alternative is a benchmark program that has been specifically written to be similar to the application program. What is needed is a program with the similar computations, similar input/output characteristics, a similar number of tasks, and similar interactions among the tasks. This might be considered a tailored synthetic benchmark program.

The advantage of this type of benchmark is that it is more characteristic of the workload to be performed than any other benchmark. The disadvantage is that it may have to be constructed from scratch which may be a more difficult task than using something that is already written. Furthermore, these benchmarks, are subject to the same disadvantages as the composite and synthetic benchmarks, namely that they do not clearly identify sources of performance problems.

8.2. Factors Causing Variation in Results

Benchmark results may vary significantly depending on the hardware and software environment in which they are run. Evaluators should be cognizant of the various sources of variation and try to control them. They should also gain an appreciation for the magnitude of the errors introduced by these sources of variation.

Benchmark programs often yield inconsistent or inexplicable results. These variations result from the environment in which the benchmarks are run. Many of the environmental factors may be controlled, but many others are difficult or impossible to control. Fine-grained benchmarks are usually more susceptible to variation than coarse-grained benchmarks because small changes in the environment can cause large changes in the results. This section will simply enumerate some of the factors that cause difficulty either in comparing two systems, or in achieving repeatability on a single system. The reader is referred to papers by Altman [Altman 87], Clapp, et al. [Clapp 86], and Gentleman, et al. [Gentleman 73] for further details.

- **Memory effects:**
  - Cycle stealing—Peripherals or other processors may "steal" memory cycles and slow down the processor speed.
  - Boundary alignment—Segments of code that are spread across memory boundaries may run more slowly than segments of code that are fortuitously aligned in memory.
  - Memory interleaving—Whether a double word operand is located on an even-odd or odd-even location may make a difference.
  - Multi-level memories—Cache, scratch pad, and paged memories operate at different speeds and affect performance.
- **Processor effects:**
  - Pipelined architectures—Instruction look-ahead and overlap can influence the execution speed of instruction sequences.
  - Interrupts—Interrupts and interrupt service routines can influence performance.
  - Clocks—Hardware clocks vary in their timing resolution.
- **Operating and runtime system effects:**
  - General overhead—The operating system may require processor time to allocate and manage system resources such as the clock or memory.
  - Periodic and asynchronous events—Operating system "daemons" to

handle events such as network activity may be activated at unpredictable times and steal processor time.

- Garbage collection—The Ada runtime system may collect unused memory at unpredictable times.

- Multiprogramming—Other programs executing will influence the elapsed time and possibly the CPU time.

- **Program translation effects:**

  - Optimization—Variation is caused when parts of the benchmark are optimized away by the compiler at compile time or link time.

  - Asymetrical translation—A compiler may translate the first instance of a language construct differently from second and subsequent instances.

  - Hidden parallelism—I/O from compilation may be performed in parallel with execution of a test.

8.3. Timing Anomalies

Timing methodology is crucial to reliable benchmarking results. Most Ada benchmark suites rely on software techniques for timing. Package CALENDAR provides an implementation-dependent timing capability so that timing issues must be carefully studied and understood.

Most benchmark programs use the Ada package CALENDAR rather than external timing mechanisms or operating system timing mechanisms. The first possible problem for benchmarking is the lack of precision of this clock. Many Ada implementations have a clock resolution of 10 milliseconds or greater. This is not sufficient for the fine-grained benchmarks described above. The solution to this problem is to use a "dual loop design." In this technique, the software to be timed is repeated many times in a loop and then the overhead of the loop is subtracted by timing a "control loop" [Clapp 86]. This technique can be refined by a "software vernier" which provides additional precision [Wright Research and Development Center 88c]. Another possible problem is that the software implementing package CALENDAR may be subject to some of the factors causing variation that are listed in the previous section.

The dual loop design depends heavily on the assumption that the loop instructions in the test loop take exactly the same amount of time as the loop instructions in the control loop. Unfortunately, this may not always be the case because of the variations described in the previous section (particularly memory alignment variations). Because the dual loop design requires the subtraction of two large numbers of nearly the same value, a small relative error in either of the numbers may cause a large relative error in the difference. The problems of dual loop benchmarks are fully documented [Altman 88]. In its worst manifestation, this

anomaly will result in negative values being provided for the time required to execute certain features of the language.

8.4. Timing Verification

> *Software timing techniques are not always reliable. Two techniques are available to verify software timings. The first is to use a high precision hardware clock with an Ada interface package. The second is to use direct hardware monitoring with, for example, a logic analyzer or in-circuit emulator.*

Many of the sources of timing variation mentioned in Section 8.2 are not introduced by software timing techniques but are *real* effects that modify the time it takes to execute instruction sequences being measured. These variations due to real effects will be measured by both hardware and software techniques. To isolate software timing variations, software timing results should be verified with hardware when time and resources permit. Hardware clocks are generally at least three orders of magnitude more precise than software timers. Whereas the Ada package CALENDAR may provide a timer resolution of 10 milliseconds, hardware clocks may provide a timer resolution of 1 microsecond. This permits the user to time a language construct directly instead of using the dual loop design. It also allows the user to measure the distribution of the times required to execute a language construct rather than just computing an average. One would hope that it always takes the same amount of time to execute an Ada language feature, but because of operating system and runtime system effects, this may not be the case. In order to use a hardware clock, the user must purchase the timer hardware (if it is not already on the processor board) and write the drivers to start and read the clock. One model for interfacing Ada with a high precision timer is given by Borger [Borger 87].

A second method of verifying timing data is to use hardware monitoring devices. Both logic analyzers and microprocessor development systems (MDSs) with in-circuit emulators (ICEs) provide capabilities that are helpful to real-time embedded programmers. They differ in cost and sophistication. A logic analyzer is a monitoring and storage device while an MDS is a programmable device that can, with an ICE, be used to both monitor and control the activities of the target system within its embedded environment. These devices are becoming increasingly more flexible and sophisticated; they can be used to debug on the target using Ada source code.

A logic analyzer can be used to store samples of data from a hardware source such as an address bus. It can be triggered to start and stop sampling using a variety of criteria. In particular, it can be triggered to take a sample once a certain address appears on the address bus. Once the data has been captured, the timing characteristics can usually be measured down to the nearest 50 nanoseconds or better. The logic analyzer permits the user to compute times that are impossible to determine using software techniques. For example, no software technique can be used to determine the time interval between an interrupt signal and the start of the interrupt service routine. Thus, for interrupt handling and real-time programming and debugging, the logic analyzer or MDS can be an indispensable tool.

8.5. Data Analysis and Reporting

> *When using suites of benchmark programs, the user may be faced with huge amounts of data. It is extremely useful to have programs that analyze and display the data in a helpful fashion, but these tools cannot substitute for sound engineering analysis. This is particularly important in discovering the weakest and strongest points of each compiler.* Some of the benchmark test suites discussed

in Chapter 9 have as many as 1000 individual tests, each providing the time and space required to execute the test. Obviously, this presents a formidable data reduction task. The numbers by themselves represent an overwhelming quantity of data and a paucity of information. What is extremely helpful is some means of making some sense of the information, especially for comparing one system with another system. In particular, it is useful to know the overall performance of a compilation system compared to that of other compilation systems, as well as results from tests that fall outside nominal expectations compared to test results of other systems. The ACEC provides some of this kind of software.

Good analysis tools make it is easier to conduct sound engineering analysis of the tradeoffs involved in the selection and use of compilers. The analysis must bring out the fact that there is no single rank-ordered list of compilers for a given application. Each Ada compiler offers something different and should be judged on the basis of its strengths and weaknesses rather than against a rigid set of criteria. For example, a compilation system that scores highly on most criteria and has only one fatal flaw may be unacceptable compared to a compiler that scores less highly on all criteria. Test suites and analysis software may give the impression that evaluation is based on apples-to-apples comparisons, when in fact the compilation systems being compared are not all addressing the same problem and should not all be judged using the same yardsticks.

8.6. Strategy for Benchmarking

> *Benchmarking should be done very selectively. Because so many programs are available, planning is necessary to choose the most salient benchmarks for a particular application. Most times the user is well advised to augment the selected benchmarks with tailored application profiles that will more closely represent the application than any of the publicly available benchmarks.*

In their paper, "Computer Benchmarking: Paths and Pitfalls," Dongarra et al. [Dongarra 87] provide the following advice: "If a performance evaluation is to be effective, it will include:

- Accurate characterization of the workload.
- Initial tests using simple programs.
- Further tests with programs that approximate even more closely the jobs that are part of the workday."

While the primary concern of their paper is benchmarking hardware rather than software

systems, the advice is still relevant for Ada compilations systems. It should be recognized that benchmarking is only one aspect of compiler system evaluation and should not be the only criterion.

8.7. Standard Benchmark Configuration Information

> *Certain key information should be provided whenever benchmark figures are presented. If any of the information is missing, the user of the benchmark data should request it. If the information cannot be provided, the consumer should treat the results with extreme skepticism.*

The following information on standard benchmark configurations includes both compile-time and execution-time benchmarks:

- For the host system (for compile-time benchmarks):
  - the host machine (including model number, memory size, memory speed)
  - the peripherals (disk type, capacity, interface, and speed)
  - the operating system (including version number)
  - relevant configuration parameters for host operating system
- For the target system (for runtime benchmarks):
  - the target configuration (including number of processors, cycle speeds, memory sizes, cache sizes, existence of floating point co-processors, number of wait states for the memory)
  - the target operating system (if any, including version number)
- For all benchmarks:
  - compiler system vendor
  - time and date of test
  - version number of the compiler *and* runtime system
  - switch settings for compilations
  - option settings for linking/loading
  - test suite version number
  - test name
  - list of all modifications made to test
  - list of all special environmental considerations (network interfaces disabled, daemons disabled, other processing loads, co-processor enabled/disabled, etc.)

- information on the timing mechanism and units of measurement (e.g., CPU time or elapsed wall clock time)

In short, the supplier of benchmark data should provide all the information that is necessary to reproduce exactly the same results using exactly the same configuration. If this condition is not met, then it is not safe to assume any particular configuration information. The benchmark results should be considered suspect until independently verified and documented.

9. Test Suites and Other Available Technology

There is no single appropriate test suite or checklist for all possible uses of an Ada compilation system. Different technology tests different aspects of a compilation system. Some test suites place more emphasis on runtime performance while others may place more emphasis on the support tools provided with a compilation system. However, it will rarely be the case that an evaluator will have to start from scratch. Much technology exists for evaluating the quality of Ada compilation systems. It should be noted, however, that for embedded systems there is little automation and gathering results may be tedious and time-consuming.

9.1. General Information on Evaluation and Test Suites

The generic issues of benchmarking are discussed in Chapter 8. This chapter gives brief descriptions of some of the existing benchmarking technology along with its strengths and weaknesses. In general, the technology described in this chapter has the following characteristics:

- There are more fine-grained language feature tests than there are composite or synthetic benchmarks.
- The fine-grained tests are subject to the timing anomalies described in Chapter 8.
- The tests have tended to address portability to a larger extent than real-time performance.
- With a few exceptions, the test suites provide little or no analysis capability.
- None has any graphical output of comparative results, with the exception of the printer histogram plots of the ACEC. In short, there are many Ada test programs, many checklists, and many test suites, but very little in the way of advice for applying this technology and almost nothing in the way of analyzing vast quantities of raw data.

The remainder of the chapter focuses on five of the better known benchmark test suites (ACEC, PIWG, AES, University of Michigan, and Aerospace), one software repository (ASR), and other sources of evaluation information. For each of these, the major strengths and weaknesses of the technology involved are noted. For summaries of each of the test suites as described by their own documentation, see Appendix A.

9.2. The Ada Compiler Evaluation Capability

The ACEC is the test suite developed by the U.S. government for evaluating compiler systems. It has been available from the government since September 1988. The test suite will be expanded over the next year. **Primary strengths:** *extensive coverage of language features and analysis tools.* **Primary weaknesses:** *relative newness and lack of support.*

The Ada Compiler Evaluation Capability (ACEC) is a comprehensive test suite for assessing the performance characteristics of Ada compilers. It was developed by Boeing Military Airplanes for the APSE Evaluation and Validation (E&V) Team of the Ada Joint Program Office. According to the *ACEC User's Guide* [Wright Research and Development Center 88d], "the ACEC shall make it possible to:

1. Compare the performance of several implementations. The Operational Software shall permit the determination of which is the better performing system for given expected Ada usage.

2. Isolate the strong and weak points of a specific system, relative to others which have been tested. Weak points, once isolated, can be enhanced by implementors or avoided by programmers.

3. Determine what significant changes were made between releases of a compilation system.

4. Predict performance of alternate coding styles. For example, the performance of rendezvous may be such that designers will avoid tasking in their applications. The ACEC will provide information to permit users to make such decisions in an informed manner."

The ACEC consists of 240 test programs comprising over 1000 tests and some support tools to analyze test results. The main emphasis of the ACEC is execution performance, but it also addresses compile-time efficiency and code-size efficiency. The test suite includes:

- language feature tests
- composite benchmarks
- optimization tests
- sorting programs
- example avionics application

Files of raw output from the test programs can be formatted by a support program named FORMAT and then used as input to an analysis tool called MEDIAN. This program can be used to perform a statistical analysis of the ACEC results collected from several target systems. The output of MEDIAN takes the form of statistical summaries and histograms which can be used to compare the performance of different target systems.

The ACEC has been available from the Data Analysis Center for Software (DACS) since September 1988. Sample command files for VAX/VMS and UNIX systems are provided with the current release; users will have to adapt these if they wish to run the ACEC on other host machines. The principal documents describing the ACEC are the *Reader's Guide* [Wright Research and Development Center 88c], the *User's Guide* [Wright Research and Development Center 88d], and *Version Description Document* [Wright Research and Development Center 88e].

For a more complete description of the ACEC, the issues involved in using it, and the taxonomy of coverage, the reader is referred to a series of questions and answers published in the *Ada-JOVIAL Newsletter* [Lange 88].

As of this writing, the current release of ACEC is Version 1.0. Plans for the next version (subject to government funding availability) include adding new tests in response to user feedback and evaluating the quality of diagnostic messages, debuggers, and library system. There is also a desire to provide more analysis capabilities for single systems.

- **Strengths:**
  - depth of coverage for language features and runtime optimization (goes well beyond that of other test suites)
  - well documented structure and taxonomy
  - extensive documentation
  - code size measurements
  - good timing techniques with statistical model and indication of accuracy
  - cross-system analysis software
  - expectation of further development
- **Weaknesses:**
  - no support provided by government or contractor (at present)
  - need for further automation in analysis subsystem
  - weakness of first version's systematic coverage of compile-time performance
  - first version's lack of coverage of diagnostics, debuggers, or library system
  - export control—available only to qualified DoD contractors

The ACEC goes well beyond the PIWG and Michigan suites in its range of tests, its timing techniques (described in the *User's Guide*), and its provision of an analysis tool for results. In particular, the MEDIAN analysis program can compare ACEC test results from different machines. Furthermore, it contains a better sampling of coarse-grained MCCR application benchmarks than do either of the other two suites.

9.3. The PIWG Benchmarks

> *The PIWG benchmarks are constructed and maintained by a volunteer subgroup of SIGAda. They have been in the public domain since 1986, are updated periodically, and are widely quoted by compiler vendors.* **Primary strength:** *wide distribution and availability.* **Primary weakness:** *lack of documentation and support.*

The PIWG benchmarks are a suite of Ada performance measurement programs put together by the Performance Issues Working Group (PIWG) of the Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda). The principal focus of the tests is the measurement of the execution time of individual features of the Ada language. Examples of the kinds of tests in the PIWG suite are:

- clock resolution
- task creation and task rendezvous
- dynamic storage allocation
- exception handling
- representation clauses and operations on packed and unpacked arrays
- procedure calling overhead
- runtime checks overhead
- composite benchmarks (including Whetstone and Dhrystone)
- Hennessy benchmarks
- compilation speed and capacity tests

There are 136 tests in the suite, plus command files for running the tests. Tests are designed to be as machine-independent as possible and to run without modification. Machine-dependent CPU time routines are available for several implementations. Many of the tests are adaptations of the Ada benchmarks developed at the University of Michigan. A user can adapt the command files to run selected groups of tests and route the output to an output file. Each test program will print out a short description of the test and the measured

execution time. The only documentation currently available is the "read me" file provided with the suite.

The PIWG suite is largely the work of dedicated volunteers, so enhancements and additions do not necessarily occur on a predictable basis. A new version of the suite is generally released once a year; as of this writing, the currently available version is designated TAPE_12_12_87. The principal distribution media are tapes and diskettes, but the suite is also available from the Ada Software Repository (see Section 9.7). Users who obtain the PIWG suite are encouraged to run the tests and submit the results to PIWG; they are also encouraged to suggest enhancements or contribute new tests. A PIWG workshop is held annually and there are PIWG meetings held during the year, usually in conjunction with scheduled SIGAda meetings. There is also a PIWG newsletter which publishes PIWG test results submitted by users. PIWG activities are announced in *Ada Letters* and each issue lists the names and addresses of current PIWG officers. Contact information for PIWG is contained in Appendix B.

The PIWG suite is emerging as a kind of industry standard. It has become a generally accepted means of providing a good first cut at Ada real-time performance measurement. The tests can easily be run on systems with text I/O capabilities. As more users acquire and run the tests, a large database of results is being accumulated by PIWG. A special issue of *Ada Letters* will contain a rationale for the design of the PIWG suite. Results based on approximately 150 test reports will be presented. A PIWG database of test reports is under development.

- **Strengths:**
  - widely distributed and used (becoming an industry standard)
  - not time-consuming to run
  - available online for those with ARPANET access
  - free

Weaknesses:

- little documentation and support
- focus mostly on fine-grained runtime performance tests
- lack of analysis tools
- lack of coverage of diagnostics, debuggers, or library system

9.4. The Ada Evaluation System

The AES is the test suite for compilation system evaluation developed by the British government. It has been available since September 1987. Further development of the test suite is in progress. **Primary advantages:** *broad coverage of compilation system issues, including checklists.* **Primary** *disadvantages: cost and lack of analysis tools.*

The Ada Evaluation System (AES) is a comprehensive, automated test suite for evaluating various aspects of a minimal Ada Programming Support Environment (APSE), where a "minimal" APSE means one that contains an Ada compiler and program library system, a linker, a loader, a symbolic debugger, and runtime libraries. The test suite measures such features as:

- compile-time and execution-time performance
- quality of the generated code
- quality of the error and warning messages produced by the compiler, linker, and program library system
- capabilities of the debugging system

The AES was developed by Software Sciences Ltd. for the United Kingdom Ministry of Defence. It is distributed by the Information Technology Department of the British Standards Institute. The software consists of a suite of test programs and a test harness to run the tests. The test harness allows a user to run all or portions of the test suite and have reports on the results generated automatically. There are over 200 tests grouped into 19 main categories; some examples of these groups are:

- Group A - Compiler performance tests
- Group G - Compiler capacity tests
- Group L - General tasking tests
- Group O - Optimization tests

- Group Q - Runtime limit tests
- Group V - Benchmark tests
- Group W - Symbolic debugger tests

For evaluation of aspects of a compilation system not readily subject to automation (e.g., quality of diagnostic and informational messages), evaluation checklists and assessment guidelines are provided for users.

Two versions of the AES are available: one for DEC VAX/VMS systems and one for IBM MVS systems. The AES can be used to evaluate any compiler or cross-compiler hosted on these machines. It can also be re-hosted on other systems; the AES documentation contains re-hosting guidelines for users. Documentation comprising five user manuals is supplied on the distribution tape. The manuals are: *User Introduction* (2 volumes), *User Guide*, *Re-Hosting Guide*, and *Installation Guide*. Paper copies of the *Installation Guide* and a list of the latest fixed and outstanding bugs are included with the distribution tape.

The British Standards Institute aims to provide a comprehensive Ada evaluation service based on the AES. Users can buy a simplified version of the AES for about $1,800 and do their own evaluation of a compilation system. As an alternative, for about $21,600 BSI will do a full evaluation using their Assessor Support System version of the AES. BSI will also undertake formal evaluations of compilation systems and issue reports. Reports may be obtained by subscribing to the evaluation service (about $3,600 annually, for 12 reports). Single copies may also be purchased for about $450.

Further development work on the AES is planned by the British Ministry of Defence. The areas to be covered by this work are still under consideration. For more information on the AES the reader is referred to the British Standards Institute (see Appendix B).

As of this writing, the currently available AES is Release A, Version 1.22.

- **Strengths:**
  - breadth of coverage (more than just runtime efficiency)
  - interactive user interface
  - automatic generation of reports from test results
  - extensive documentation
  - macro capability for test generation
  - checklists for diagnostics, library systems, vendor evaluation, etc.
  - availability of example evaluation reports

- **Weaknesses:**
    - requirement of considerable setup time
    - lack of U.S. support at present
    - depth of coverage in runtime performance
    - subjective nature of checklists
    - cost

9.5. The University of Michigan Ada Benchmarks

The University of Michigan was among the first to attempt to put Ada benchmarking on a sound scientific and theoretical basis. Their CACM article is excellent reading for benchmarking practitioners. The University of Michigan work has been largely superseded by the PIWG activities described in Section 9.3. **Primary strength:** *timing mechanism based on sound theoretical principles.* **Primary weakness:** *no further development or support.*

The University of Michigan Ada benchmarks concentrate almost exclusively on the runtime performance of individual Ada language features. The suite contains over 150 different test programs to measure the execution times of task creation and rendezvous, clock calling overhead, procedure calling overhead, exception handling, and dynamic storage allocation. There is no composite benchmark, since the approach adopted by the Michigan team was to develop a *set* of benchmarks rather than one or two synthetic composites. Also, because the emphasis is on runtime performance, there are no compilation speed or capacity tests. The suite is largely similar to the PIWG suite and in fact forms the basis of much of the PIWG suite. It also contains some tests not included in the PIWG suite, such as tests for the presence of garbage collection and manipulation of variables of type **time** and **duration**.

The only documentation available on the Michigan suite is the August 1986 CACM article by Clapp et al [Clapp 86], which describes the rationale for the tests, summaries of their operation, and the theoretical basis of the measurement techniques used in the tests. Sample results for a number of machines and analyses of the results are also included in the article. There is also a follow-up letter in the February 1987 issue of CACM. For further contact information see Appendix B.

As of this writing, the currently available version of the Michigan suite is the original one described in the CACM article.

The Michigan suite is not as streamlined and flexible as the PIWG suite, probably because it was not originally designed with wide distribution in mind. However, the tape containing the suite does contain command files for compiling, linking and running the tests. Users may find it easier to use PIWG, supplemented with the Michigan tests that are not included in the PIWG suite.

- **Strengths:**
  - excellent theoretical underpinnings
  - still many requests by evaluators
- **Weaknesses:**
  - lack of current support by the University of Michigan
  - lack of further development activity

- lack of analysis tools

- lack of coverage of diagnostics, debuggers, or library system

9.6. Aerospace Benchmarks

The Aerospace Corporation has recently completed a suite of tests and an associated user's guide. The tests are written in three languages (Ada, JOVIAL, and FORTRAN). Another report gives detailed results and analyses comparing Ada with JOVIAL. **Primary strength:** *only test suite that allows comparison of language implementations.* **Primary weakness:** *availability and support still under consideration.*

The purpose of Ada Compiler Performance Test Suite and Test Evaluation Capability (ACPS) [Kayfes 88] is to "assist users in evaluating the performance of runtime environments provided by Ada compilation systems." The test suite contains feature tests as well as composite tests. A unique feature of this suite is that there are Ada, JOVIAL, and FORTRAN files so that the three languages can be compared on machines that support these languages. The suite also provides software to "gather and report performance statistics in a format common to all three test languages." The suite can be used to gather compile-time and execution-time statistical information such as elapsed time, CPU time, code and data size, virtual and physical memory usage, and the like. The base Ada test suite is comprised of 868 tests in 459 programs.

A draft Aerospace report compares two Ada compilers, a FORTRAN compiler, and a JOVIAL compiler using the ACPS [Byrne 88] on the same machine architecture. Results are presented using histograms showing the number of tests that are faster by a certain percent in each language implementation. The study showed that for those language features that are available in each language, VAX Ada was faster than VAX FORTRAN in about as many cases as FORTRAN was faster than Ada. Among the reasons cited for Ada tests running faster were automatic in-lining of procedures by the Ada compiler and the use of the machine architecture for handling slices with single instructions rather than loops. The Ada compiler was cited as slower in tests using the compiler's inefficient mechanism for accessing global variables. For the most part, the histograms showed bell-shaped curves indicating that as the ratio of execution times increased there were fewer tests in that category. However, there were a substantial number of tests for which an Ada test ran more than five time slower or a FORTRAN test ran five times slower than its equivalent. The reasons for these anomalies must be very carefully examined by evaluators interested in cross-language analysis.

- **Strengths:**

  - some tests written in three languages (Ada, JOVIAL, and FORTRAN)

  - well documented structure and taxonomy

- some analysis software available

- **Weaknesses:**
  - still uncertain in availability and support
  - written primarily for VAX/VMS-hosted systems
  - relatively new and untested

9.7. Ada Software Repository

The Ada Software Repository is a store of Ada programs, software components, and educational material. The programs include a number of benchmarks and Ada programs that could be used for evaluation purposes. **Primary strength:** *everything available online at no cost.* **Primary weakness:** *quality of programs and documentation is highly variable, depending on the source.*

The Ada Software Repository (ASR) was established in 1984. It resides on the SIMTEL20 host computer on the Defense Data Network. It contains Ada programs, software components, and educational material. The information in the repository is extremely well organized and its directory structure and files can be scanned and transferred by employing the file transfer protocol (ftp) program on a remote host system. The ASR is also available on magtape, floppy disk, and CD-ROM.

The purpose of the ASR is to promote the exchange and use of Ada programs and tools and to promote Ada education by providing working examples of programs in source form for people to study and modify. The only restrictions that apply to the access and use of the software is that which is contained in the prologs of each of the programs.

Two references are essential to anyone wishing to access the ASR. The first is *The Ada Software Repository and the Defense Data Network* by Richard Conn [Conn87 87]. This handbook contains useful information about the terminology and use of the computer networks and repositories accessible from the networks. The second reference is the *Ada Software Repository Master Index* [CONN 88]. This is a looseleaf notebook which contains, for each item of software, an abstract, information on the host compiler and operating system and target environment, and a listing of all associated files and location in the ASR.

Of interest to evaluators is the chapter on benchmarks. Contained in the current ASR are two PIWG tapes containing the 5/1/87 and 8/31/86 versions of the test suite, as well as other benchmarks for capacity, language comparisons, and tasking. The repository contains general purpose Ada components, database management programs, graphics programs, and numerous programming tools. It also contains machine-readable copies of many references of interest to evaluators such as the *RM*, *Ada Adoption Handbook*, and various ARTEWG documents.

- **Strengths:**
    - large source of Ada information and Ada programs
    - well documented structure and taxonomy
    - online availability to ARPANET users
    - no cost

Weaknesses:

- little quality control over submitted items
- no support for software
- some out-of-date information

9.8. Other Sources

Other private organizations are in the business of selling evaluation technology in the form of reports or benchmark tests. **Primary strength:** *value may be added to otherwise available data.* **Primary weakness:** *these services may be costly.*

Several small companies provide newsletters or other digests of Ada information. They include Grebyn Corporation (*INFO-Ada Newsletter*, Ada commentaries, machine-readable forms of various data), Cutter Information Corporation (*Ada Strategies Newsletter*, guidebook), and International Resource Development, Incorporated (*Ada Data Newsletter*). See Appendix B for contact information. Among large corporations that have widely circulated Ada newsletters are Texas Instruments, Sperry, and General Dynamics.

As test suites such as the PIWG tests, University of Michigan benchmarks, and the ACEC become part of the public domain, there may be more small companies packaging these and other evaluation tools into a service which applies the tests to particular compilers and makes the raw information easier to understand and analyze. These companies may provide added value to information already available in the public domain, but the watchword must be *caveat emptor*. The consumer of this information must be confident that the producer is fully cognizant of the principles that are provided in this handbook and has no vested interests in the products being evaluated.

References

[Ada Runtime Environment Working Group 87]
Ada Runtime Environment Working Group.
Catalog of Ada Runtime Implementation Dependencies.
Technical Report, ACM Special Interest Group on Ada, 11 West 42nd St., New York, NY 10036, December, 1987.

[Ada Runtime Environment Working Group 88a]
Ada Runtime Environment Working Group.
A Framework for Describing Ada Runtime Environments.
Ada Letters 8(3):51-68, May/June, 1988.

[Ada Runtime Environment Working Group 88b]
Ada Runtime Environment Working Group.
Catalog of Interface Features and Options.
Technical Report, ACM Special Interest Group on Ada, 11 West 42nd St., New York, NY 10036, October, 1988.

[Altman 87] N. Altman.
Factors Causing Unexpected Variation in Ada Benchmarks.
Technical Report CMU/SEI-87-TR-22, ADA187231, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, October, 1987.

[Altman 88] N. Altman and N.H. Weiderman.
Timing Variation in Dual Loop Benchmarks.
Ada Letters 8(3):98-102, May/June, 1988.

[Borger 87] M.W. Borger.
VAXELN Experimentation: Programming a Real-Time Clock and Interrupt Handling Using VAXELN Ada 1.1.
Technical Report CMU/SEI-87-TR-32, ADA200612, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, October, 1987.

[Bryan 88] D. Bryan.
Dear Ada.
Ada Letters 8(4):24-34, July/August, 1988.

[Byrne 88] D.J. Byrne.
Comparison of Ada Real-Time/Run-Time Environments Using the ACPS.
Technical Report SD-TR-88 (draft), Space Division, Air Force Systems Command, Los Angeles Air Force Station, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009-2960, December, 1988.

[Clapp 86] R.M. Clapp, L. Duchesneau, R. Volz, T.N. Mudge, and T. Schultze.
Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 29(8):760-778, August, 1986.

[CONN 88] R. Conn (ed.).
Ada Software Repository (ASR) Master Index.
24 May edition, Management Assistance Corporation of America, PO Drawer 100, Building T-148, White Sands Missile Range, New Mexico 88002, 1988.

[Conn87 87] R. Conn.
The Ada Software Repository and the Defense Data Network.
New York Zoetrope, 838 Broadway, New York, NY 10003, 1987.

[Curnow 76] H.J. Curnow and B. A. Wichmann.
A Synthetic Benchmark.
Computer Journal 19(1):43-49, January, 1976.

[Dijkstra 72] E.W. Dijkstra.
Notes on Structured Programming.
Structured Programming.
Academic Press, New York, 1972.

[Dongarra 87] J. Dongarra, J.L. Martin, and J. Worlton.
Computer Benchmarking: Paths and Pitfalls.
IEEE Spectrum 24(7):38-43, July, 1987.

[Feiler 88] P.H. Feiler and R. Smeaton.
Managing Development of Very Large Systems: Implications for Integrated Environment Architectures.
Technical Report CMU/SEI-88-TR-11, ADA197671, Software Engineering Institute, Carnegie Mellon University, May, 1988.

[Firesmith 88] D. G. Firesmith.
Mixing Apples and Oranges or What is an Ada Line of Code Anyway?
Ada Letters 8(5):110-112, September/October, 1988.

[Firth 88] R. Firth.
Ada Code Quality, Some Observations and Some Recommendations.
Presentation at the AdaJUG Conference in Redondo Beach, CA.
December 1, 1988.

[Fleming 86] P.J. Fleming and J.J. Wallace.
How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results.
Communications of the ACM 29(3):218-221, March, 1986.

[Foreman 87] J.T. Foreman and J.B. Goodenough.
Ada Adoption Handbook: A Program Manager's Guide.
Technical Report CMU/SEI-87-TR-9, ADA182023, Software Engineering Institute, Carnegie Mellon University, May, 1987.

[Ganapathi 89] M. Ganapathi and G.O. Mendal.
Issues in Ada Compiler Technology.
IEEE Computer 22(2):52-60, February, 1989.

[Gentleman 73] W.M. Gentleman and B.A. Wichmann.
Timing on Computers.
Computer Architecture News 2(3):20-23, October, 1973.

[Goodenough 81] J.B. Goodenough.
The Ada Compiler Validation Capability.
IEEE Computer 14(6):57-64, June, 1981.

[Goodenough 86a] J.B. Goodenough.
An Example of Software Testing Theory and Practice.
System Development and Ada, Proceedings of the CRAI Workshop on Software Factories and Ada, Capri, Italy, May 1986.
Springer-Verlag, 1986, pages 195-232.

[Goodenough 86b] J. B. Goodenough.
Ada Compiler Validation Capability Implementers' Guide.
Technical Report NTIS ADA189647, SofTech, Inc., Waltham, MA, December, 1986.

[Hogan 87] M.O. Hogan, E.P. Hauser, and S.M. Menichiello.
The Definition of a Production Quality Ada Compiler.
Technical Report SD-TR-87-29, Space Division, Air Force Systems Command, Los Angeles Air Force Station, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009-2960, March, 1987.

[Ichbiah 86] J.D. Ichbiah, J.G.P. Barnes, R.J. Firth, and M. Woodger.
Rationale for the Design of the Ada Programming Language.
Ada Joint Program Office, OUSDRE(R&AT), The Pentagon, Washington, D.C. 20301, 1986.

[Kayfes 88] R.E. Kayfes.
Ada Compiler Performance Test Suite and Test Evaluation Capability (ACPS) User's Guide.
Technical Report SD-TR-88 (draft), Space Division, Air Force Systems Command, Los Angeles Air Force Station, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009-2960, June, 1988.

[Lange 88] D. Lange.
Ada Compiler Evaluation Capability (ACEC) Questions and Answers, Version 1.0 --- 9/3/88.
Ada-JOVIAL Newsletter 10(3):16-20, September, 1988.

[Lyons 86] T.G.L. Lyons and J.C. D. Nissen.
Selecting an Ada Environment.
Cambridge University Press, New York, 1986.

[Marshall 87] I. Marshall.
Ada Evaluation System
18 June edition, British Standards Institute, Linford Wood, Milton Keynes, UK, 1987.

[Nissen 84] J.C.D. Nissen and B.A. Wichmann.
Ada-Europe Guidelines for Ada Compiler Specification and Selection.
Ada Letters 3(5):50-62, March/April, 1984.
Originally published in October 1982 as National Physical Laboratory (United Kingdom) Report DITC 10/82.

[Shepherd 88] R. Shepherd and P. Thompson.
Lies, Damned Lies and Benchmarks.
Technical Report 27, INMOS Limited, Bristol, U.K., January, 1988.

[Tetewsky 87] A. Tetewsky and R. Racine.
Ada Compiler Selection for Embedded Targets.
Ada Letters 7(5):51-62, September/October, 1987.

[U.S. Army HQ Center for Software Engineering 89]
U.S. Army HQ Center for Software Engineering.
Final Report - Guideline to Select, Configure, and Use an Ada Runtime Environment.
Technical Report CIN: C02092LA0001, CECOM Center for Software Engineering, Advanced Software Technology, Fort Monmouth, NJ, 15 February, 1989.

[United States Department of Defense 80]
United States Department of Defense.
Requirements for Ada Programming Support Environments.
Technical Report DTIC Report Number ADA100, DoD, February, 1980.

[United States Department of Defense 83]
United States Department of Defense.
Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.
American National Standards Institute, New York, 1983.

[United States Department of Defense 87a]
United States Department of Defense.
DoD Directive 3405.1, Computer Programming Language Policy.
April 2, 1987.

[United States Department of Defense 87b]
United States Department of Defense.
DoD Directive 3405.2, Use of Ada in Weapon Systems.
March 30, 1987.

[United States Department of Defense 87c]
United States Department of Defense.
Ada Compiler Validation Procedures and Guidelines.
Technical Report AD A178 154, NTIS, U.S. Dept. of Commerce, 5285 Port Royal Rd., Springfield, VA 22161, January, 1987.

[Wand 87] I.C. Wand, J.R. Firth, C.H. Forsyth, L. Tsao, and K.S. Walker.
Facts and Figures About the York Ada Compiler.
Ada Letters 7(4):85-87, July/August, 1987.

[Weicker 84] R.P. Weicker.
Dhrystone: A Synthetic Systems Programming Benchmark.
Communications of the ACM 27(10):1013-1040, October, 1984.

[Weiderman 87a] N.H. Weiderman, N. Haberman, et al.
Evaluation of Ada Environments.
Technical Report CMU/SEI-87-TR-1, ADA180905, Software Engineering Institute, Carnegie Mellon University, March, 1987.

[Weiderman 87b] N.H. Weiderman, M.W. Borger, A.L. Cappellini, S.A. Dart, M.H. Klein, and S.F. Landherr.
Ada for Embedded Systems: Issues and Questions:.
Technical Report CMU/SEI-87-TR-26, ADA191096, Software Engineering Institute, Carnegie Mellon University, December, 1987.

[Wolf 81] M.I. Wolf, W. Babich, R. Simpson, R. Thall, and L. Weissman.
The Ada Language System.
IEEE Computer 14(6):37-45, June, 1981.

[Wright Research and Development Center 88a]
Wright Research and Development Center.
E & V Reference Manual, Version 1.1.
Technical Report TASC-TR-5234-3, Wright Patterson AFB, OH 45433, (DTIC Accession Number pending), October, 1988.

[Wright Research and Development Center 88b]
Wright Research and Development Center.
E & V Guidebook, Version 1.1.
Technical Report TASC-TR-5234-4, Wright Patterson AFB, OH 45433, (DTIC Accession Number pending), August, 1988.

[Wright Research and Development Center 88c]
Wright Research and Development Center.
Ada Compiler Evaluation Capability (ACEC) Technical Operating Report (TOR) Reader's Guide.
Technical Report AFWAL-TR-88-1094, Wright Patterson AFB, OH 45433, (DTIC Accession Number pending), August, 1988.

[Wright Research and Development Center 88d]
Wright Research and Development Center.
Ada Compiler Evaluation Capability (ACEC) Technical Operating Report (TOR) User's Guide.
Technical Report AFWAL-TR-88-1095, Wright Patterson AFB, OH 45433, (DTIC Accession Number pending), August, 1988.

[Wright Research and Development Center 88e]
Wright Research and Development Center.
Ada Compiler Evaluation Capability (ACEC) Version Description Document.
Technical Report AFWAL-TR-88-1093, Wright Patterson AFB, OH 45433, (DTIC Accession Number pending), August, 1988.

Appendix A: Test Suite Summaries

A.1. ACEC Test Groupings

The following list is taken from the table of contents of the *ACEC Reader's Guide* [Wright Research and Development Center 88c]:

- Execution Time Efficiency
    - Individual Language Features
    - Pragmas
    - Optimizations
        - Classical Optimizing Techniques
            - Common Subexpression Elimination
            - Folding
            - Loop Invariant Motion
            - Strength Reduction
            - Dead Code Elimination
            - Register Allocation
            - Loop Interchange
            - Loop Fusion
            - Test Merging
            - Boolean Expression Optimization
            - Algebraic Simplification
            - Order of Expression Evaluation
            - Jump Tracing
            - Unreachable Code Elimination
            - Use of Machine Idioms

        - Packed Boolean Array Logical Operators

- Effects of Pragmas
- Static Elaboration
        - Aggregates
        - Tasks
- Language Specific
        - Habermann-Nassi Transformation For Tasking
        - DELAY Statement Optimization
- Performance Under Load
        - Task Loading
        - Levels of Nesting
        - Parameter Variation
        - Declarations
- Tradeoffs
        - Design Issues
                - Order of Evaluation
                - Default vs. Initialized Records
                - Order of Selection
                - Scope of Usage
                - LOOP Statements
                - CASE Statements
                - Subtypes
                - Generics
                - Library Subunits
                - Exceptions

- Context Variation
    - Different Coding Styles
- Operating System Efficiency
    - Tasking
    - Exception Handling
    - File I/O
    - Memory Management
    - Elaboration
    - Runtime Checks
- Application Profile Tests
    - Classical Benchmark Programs
    - Ada in Practice
    - Ideal Ada
- Code Size Efficiency
    - Code Expansion Size
    - Runtime System Size
- Compile Time Efficiency
- Tests for Existence of Language Features
- Usability
- Capacity Tests

A.2. PIWG Test Groupings

The following list is derived from the "READ.ME" file of the PIWG suite TAPE_12_12_87:

- Group A - Setup, clock resolution, Dhrystone, Whetstone, Hennessy
- Group B - Tracker algorithm

- Group C - Task creation
- Group D - Dynamic elaboration
- Group E - Exceptions
- Group F - Coding style
- Group G - Text_IO
- Group H - Chapter 13
- Group L - Loop overhead
- Group P - Procedure calls
- Group T - Task
- Group Y - DELAY
- Group Z - Compile time

A.3. AES Test Groupings

The following list is taken from the table of contents of the *AES User Introduction to the Evaluation Test Suite* [Marshall 87]:

- Group CH - The checkout tests
- Group A - Compiler efficiency tests
- Group B - Compiler informational quality tests
- Group C - Compiler error reporting tests
- Group D - Compiler error recovery tests
- Group E - Compiler warning tests
- Group F - Compiler behavioral tests
- Group G - Compiler capacity tests
- Group I - General run-time efficiency tests
- Group J - NPL test suite
- Group K - Tasking tests for Mascot systems
- Group L - Tasking tests
- Group M - Storage management tests
- Group N - Input/output tests
- Group O - Optimization tests
- Group Q - Run-time limit tests
- Group R - Implementation dependency tests
- Group U - Linker/loader error reporting tests
- Group V - Benchmark tests
- Group W - Symbolic debugger tests

A.4. University of Michigan Test Groupings

The following list is taken from the article in *Communications of the ACM* [Clapp 86]:

- Subprogram calls
- Object allocation
- Exceptions
- Task elaboration, activation, and termination
- Task synchronization
- CLOCK evaluation
- TIME and DURATION evaluation
- DELAY function and scheduling
- Object deallocation and garbage collection
- Interrupt response time

A.5. ACPS Test Groupings

The following list is taken from an Aerospace draft report comparing Ada real-time/runtime environments using the ACPS [Byrne 88]:

- General Tests
- Feature Tests
    - Computational Tests
        - Integer Tests
            - Integer Assignment
            - Integer Math
            - Integer Declaration
        - Floating Point Tests
            - Floating Point Assignment
            - Floating Point Math

- Fixed Point Tests

- Input/Output
  - Direct File Input/Output
  - Sequential File Input/Output
- Control Statements
  - Boolean Expressions
  - Branching
- Non-Numerical Data Processing
  - Strings
  - Record Types
    - Record Assignment
    - Record Component Reference
    - Record Conversion
  - Access Types
  - Enumeration Types
- Procedure Metrics
  - Local Calls
  - External Calls
  - Argument Reference
  - In-line Local Calls
  - In-line External Calls
  - In-line Argument Reference
  - Variable Reference
- Tasking Metrics

- Generics Metrics
- Exception Metrics
- Miscellaneous
    - Numeric Conversion
    - Other

- Loading Tests
    - Tasking I/O
    - Tasking Array Access

- Optimization Tests
    - Common Subexpression
    - Loop Optimization
    - Other Optimization

Appendix B: Compiler Evaluation Points of Contact

Many groups are involved in Ada compiler evaluation and selection activities. In the following sections, details of various groups are discussed, including:

- Professional organizations
- U.S. government sponsored/endorsed organizations
- Ada information sources

Contact information is provided; where known, AUTOVON numbers (AV) and/or electronic mail (Email) addresses are also included.

B.1. Professional Organizations

B.1.1. Ada Joint Users Group (AdaJUG)

AdaJUG is a national organization (formerly known as the Ada-JOVIAL Users Group) providing a forum for communication among persons involved with the acquisition, development, and maintenance of real-time embedded systems using Ada (and JOVIAL/J73). The AdaJUG makes recommendations to appropriate military services and DoD agencies regarding language policies and practices. Two AdaJUG points of contact are:

Mr. Joe Dangerfield, Chair
TeleSoft Corporation
5959 Cornerstone Court West
San Diego, CA 92121-9891
(619) 457-2700

Mr. Dudrey Smith
Smiths Industries, Chair, Ada Validation WG
Aerospace & Defense Systems
SLI Avionic Systems Corp.
4141 Eastern Ave, S.E.
Grand Rapids, MI 49518
(616) 241-7665

B.1.2. SIGAda

The Association for Computing Machinery Special Interest Group on Ada is a professional association composed of people interested in the Ada language. *Ada Letters* is the SIGAda bimonthly publication. The SIGAda chairman is:

Dr. Ben Brosgol
Alsys, Inc.
1432 Main Street
Waltham, Ma 02154
(617) 890-0030
Email: brosgol@ajpo.sei.cmu.edu

The following SIGAda groups are examining issues of particular interest:

ARTEWG: Ada Run-Time Environment Working Group

- **Purpose**: To establish conventions, criteria, and guidelines for Ada runtime environments that facilitate the reusability and transportability of Ada program components; improve the performance of those components; and provide a framework which can be used to evaluate Ada runtime systems [Ada Runtime Environment Working Group 87, Ada Runtime Environment Working Group 88a, Ada Runtime Environment Working Group 88b]. ARTEWG acts as a forum for users to interact effectively with Ada implementors, thereby encouraging development of runtime environments that meet users' needs. For further information, contact:

- Mr. Mike Kamrad
  Unisys Computer System Division
  M/S Y41A6
  PO Box 64525
  St. Paul, MN 55164-0525
  (612) 456-7315
  Email: mkamrad@ajpo.sei.cmu.edu

PIWG: Performance Issues Working Group

- **Purpose**: To investigate Ada compiler performance issues. Develops benchmark tests in areas such as exception handling, loop overhead, procedure calls, I/O, dynamic allocation, task creation, and task rendezvous; and collects and disseminates results. The PIWG tests have been run against many Ada compilers. Instructions to customize the tests for a particular compiler are included with the tests. For further information, contact:

- Dr. Daniel Roy, Chairman
  Ford Aerospace
  7375 Executive Place, Suite 400
  Seabrook, MD 20706-2257
  (301) 805-0464

B.1.3. ISO/JTC1/SC22/WG9

Working Group 9 (WG9) is the working group for Ada language standardization within the Programming Languages Subcommittee (SC22) of the Information Systems Joint Technical Committee (JTC1) of the International Standards Organization (ISO). WG9 is the international organization responsible for the Ada standard and any subsidiary standards. The working group conducts its business through subgroups, which are given work items. Currently there are four subgroups called the Ada Rapporteur Group (ARG), the Uniformity Rapporteur Group (URG), the Structured Query Language (SQL) subgroup and the Ada Numeric Packages subgroup. The ARG is responsible for responding to commentaries about the language and clarifying the meaning of the standard. The URG is addressing how to promote uniformity of Ada implementations in those cases where the standard leaves freedom to the implementors. The SQL subgroup is tasked to define a standard interface or

binding between SQL and Ada. It has not yet been determined whether this last subgroup will fall under SC22 or some other subcommittee (SC21). For further information contact:

Dr. Robert Mathis, Convenor of WG9
Software Engineering Laboratory
Contel Technology Center
12015 Lee Jackson Highway
Fairfax, VA 22033-3346
(703) 359-0203
Email: mathis@a.isi.edu

Dr. John Goodenough, Chairman of the ARG
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-6391
Email: goodenough@sei.cmu.edu

Dr. Robert B. K. Dewar, Chairman of the URG
New York University
715 Broadway
New York, NY 10012
(212) 998-3000
Email: dewar@acf2.nyu.edu

Mr. Stephen Michell, Chairman of the SQL Subgroup
Prior Data Sciences
240 Micheal Cowpland Drive
Kanata, Ontario K2M 1P6
Canada
(613) 596-7790

Mr. Gil Meyers, Chairman of the Ada Numeric Packages Subgroup
Naval Ocean Systems Center
Code 423
271 Catalina Boulevard
San Diego, CA 92152-5000
(619) 225-7401
Email: gmyers@ajpo.sei.cmu.edu

B.2. U.S. Government Sponsored/Endorsed Organizations

AJPO: Ada Joint Program Office

- **Purpose**: To oversee the total direction of the Ada program. The AJPO reports to the Deputy Undersecretary of Defense for Research and Advanced Technology (DUSDR&AT).

- For further information, contact:

  Mr. William S. Ritchie, Acting Director
  Ada Joint Program Office
  The Pentagon, Room 3E114
  Washington, D.C. 20301-3081
  (202) 694-0210
  Email: ritchie@ajpo.sei.cmu.edu

Ada Board:

- **Purpose**: A federal advisory committee, composed of compiler developers, language designers, embedded system users, and government personnel whose purpose is to provide the director of the AJPO with a balanced source of advice and information regarding the technical and policy aspects of the Ada Program. For further information, contact:

  Mr. William S. Ritchie, Acting Director
  Ada Joint Program Office
  The Pentagon, Room 3E114
  Washington, D.C. 20301-3081
  (202) 694-0210
  Email: ritchie@ajpo.sei.cmu.edu

AdaIC: Ada Information Clearinghouse director

- **Purpose**: To support the AJPO by distributing Ada-related information, including:

  - policy statements
  - lists of validated compilers
  - classes
  - conferences
  - text books
  - programs using Ada

  In addition to publishing a newsletter, an electronic bulletin board system (300/1200/2400 baud, no parity, 8 bits, 1 stop bit) is available at (202) 694-0215 and (301) 459-3865. For further information, contact:

Ada Information Clearinghouse
c/o IIT Research Institute (IITRI)
4600 Forbes Boulevard
Lanham, MD 20706-4312
(703) 685-1477 or (301) 731-8894
Email: adainfo@ajpo.sei.cmu.edu

AMO: Ada Maintenance Organization

- **Purpose**: To develop, maintain, and support the Ada Validation Suite (AVS). Additionally, the AMO supports Ada language maintenance activities. For further information, contact:

  Mr. Bobby R. Evans
  ASD/SCEL
  Wright-Patterson AFB, Ohio 45433
  (513) 255-4472
  Email: evansbr@wpafb-jalcf.arpa

AVF: Ada Validation Facility

- **Purpose**: To validate Ada compilers (giving priority to DoD-targeted compilers) and register derived compilers. AVFs currently exist in the US (2), the UK, France, and West Germany. For further information, contact:

  Mr. Bobby R. Evans
  ASD/SCEL
  Wright-Patterson AFB, Ohio 45433
  (513) 255-4472
  Email: evansbr@wpafb-jalcf.arpa

  In addition to the aforementioned activities, this organization also publishes the *Ada-JOVIAL Newsletter*. To subscribe, write to:

  ASD/SCEL
  Standard Languages and Environments Division
  Engineering Applications Directorate
  DCS/Communications - Computer Systems (ASD/SC)
  Wright Patterson AFB, Ohio 45433-6503
  (513) 255-4472/4473
  AV: 785-4472

AVO: Ada Validation Organization

- **Purpose**: The AVO is a federally funded research center directly responsible to the AJPO. AVO functions include:

  - overview development of the Ada Compiler Validation Capability (ACVC)

  - independent QA on RELEASED AVS (Ada Validation Suites)

  - resolution of disputes arising from problems in the validation process, such as investigating disputed tests and having incorrect tests withdrawn from the validation suite

  For further information, contact:

  Ms. Audrey Hook
  Institute for Defense Analyses
  1801 Beauregard Street
  Alexandria, Virginia 22311
  (703) 824-5501
  Email: ahook@ajpo.sei.cmu.edu

E&V: Evaluation and Validation Team

- **Purpose**: "The Ada community, including government, industry, and academic personnel, needs the capability to assess APSEs (Ada Programming Support Environments) and their components and to determine their conformance to applicable standards (e.g., DoD-STD-1838, the CAIS standard). The technology required to fully satisfy this need is extensive and largely unavailable; it cannot be acquired by a single government-sponsored professional society-sponsored, or private effort. The purpose of the APSE Evaluation and Validation (E&V) task is to provide a focal point for addressing the need by:

  1. Identifying and defining technology requirements,

  2. Developing selected elements of the required technology,

  3. Encouraging others to develop some elements, and

  4. Collecting information describing existing elements.

  5. This information will be made available to DoD components, other government agencies, industry and academia" [Wright Research and Development Center 88a].

For further information, contact:

Mr. Raymond Szymanski
WRDC/AAAF-3
Wright-Patterson AFB, Ohio 45433-6523
(513) 255-2446
AV: 785-2446
Email: szymansk@ajpo.sei.cmu.edu

U.S. Army CECOM: Communications-Electronics Command

- **Purpose**: The Advanced Software Technology section of the Center for Software Engineering, at the U.S. Army Communications-Electronics Command and (CECOM) at Ft. Monmouth, NJ, has a technical program concerned with Ada real-time and Ada runtime issues. Their purpose is to provide guidance for the embedded real-time Ada applications world and disseminate their results. For further information, contact:

  Mr. Edward Gallagher
  U.S. Army CECOM
  AMSEL-RD-SE-AST
  Ft. Monmonth, NJ 07703
  (412) 268-5758
  Email: egallagh@ajpo.sei.cmu.edu

B.3. Sources of Evaluation Technology

Ada Compiler Evaluation Capability
Data and Analysis Center for Software
RADC/COED
Building 101
Griffiss AFB, NY 13441-5700
(315) 336-0937

Performance Issues Working Group Benchmarks
Mr. Rob Spray
PIWG Tree
P.O. Box 850236
Richardson, TX 75085-0236
(214) 907-6640

Ada Evaluation System
British Standards Institute,
Information Technology Department
BSI Quality Assurance
PO Box 375 Linford Wood
Milton Keynes MK14 6LL
United Kingdom
Tel: 0908 220908

University of Michigan Benchmarks
Robotics Research Laboratory
University of Michigan
Ann Arbor, MI 48109

Aerospace Benchmarks
Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 90009
attn. Richard Ham
(213) 336-3438

B.4. Ada Information Sources

Data Analysis Center for Software (DACS)
RADC/COED
Bldg 101
Griffiss AFB, NY 13441
(315) 336-0937
Email: dacs@radc-multics

Defense Technical Information Center (DTIC)
Cameron Station
Alexandria, VA 22314
(202) 274-6871 (Registration section)
(703) 274-7633 (Reference section)

National Technical Information Service (NTIS)
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
(703) 487-4650 or
(202) 724-3374

Ada Software Repository Newsletter
Echelon, Inc.
885 N. San Antonio Road
Los Altos, CA 94022
(415) 948-3820
Email: ada-sw-request@simtel20.army.mil

Grebyn Corporation
P.O. Box 497
Vienna, VA 22180
(703) 281-2194
Email: products@grebyn.com

Cutter Information Corporation
1100 Massachusetts Ave
Arlington, MA 02174
(617) 648-8700

International Resource Development, Inc.
PO Box 1716
New Canaan, CT 06840
(203) 966-2525

Appendix C: Accessing Network Information

A great deal of information is available through network access. This information can be retrieved online so that it is often more up-to-date and more readily available than information available in libraries or ordered through the mail. Much of this information is on the AJPO and the SIMTEL20 machines which are both on ARPANET. Anyone that has ftp (file transfer protocol) access to these machines can access this information. The best resource on the various networks and information on how to navigate between them is given in *The Ada Software Repository and the Defense Data Network* [Conn87 87]. This appendix will merely give some simple scripts which should work for ARPANET users. Those on other networks will need to consult their system managers to determine whether they have ftp access to these ARPANET databases.

In the following scripts, lines on which the user enters information are preceded with asterisks. The commands and information that are typed by the user is shown in italics. It should be noted that "cd" is the UNIX "change directory" command. At any directory, the user may type "dir" for a listing of the files or directory entries that are contained in the current directory.

C.1. Retrieving Ada Issues

The script below shows how to retrieve an individual Ada commentary from the AJPO machine. It is possible to register to receive updates of language commentaries automatically by electronic mail. To be put on the notification list you should send Email to ada-comment@ajpo.sei.cmu.edu and request either to be notified of updates or to be sent the updates. Potential subscribers to this service are cautioned that the volume of the updates is large (approximately one megabyte per month) if they are sent the updated commentaries and its index.

The following script will retrieve all files that begin with the characters "ai-00032" in the "public/ada-comment" directory on the AJPO machine. This file happens to contain information about Ada commentary 32, "preemptive scheduling is required." An index to the Ada commentaries is contained in the files referenced as "ai-index*" where the * is a wildcard for the file suffixes. This file is quite long (approximately 200K bytes) and contains the Ada issues indexed by commentary number, reference manual number, status, and comment number.

```
*    ftp ajpo.sei.cmu.edu
     220 ajpo.sei.cmu.edu FTP server
     (Version 4.98 Wed Feb 19 18:51:48 EST 1986) ready.
*    Name (ajpo.sei.cmu.edu:): anonymous
*    Password (ajpo.sei.cmu.edu:anonymous): your name here
     331 Guest login ok, send ident as password.
     230 Guest login ok, access restrictions apply.
*    ftp> cd public/ada-comment
     200 CWD command okay.
*    ftp> mget ai-00032*
```

```
*   mget ai-00032-ra.wj? y
    200 PORT command okay.
    150 Opening dataconnection for ai-00032-ra.wj(128.237.2.47,1261)
    (2591 bytes).
    226 Transfer complete.
    2648 bytes received in 0.20 seconds (13 Kbytes/s)
*   mget ai-00032.mss? y
    200 PORT command okay.
    150 Opening data connection for ai-00032.mss (128.237.2.47,1262)
    (2595 bytes).
    226 Transfer complete.
    2660 bytes received in 0.22 seconds (12 Kbytes/s)
*   ftp> quit
    quit
    221 Goodbye.
```

C.2. Retrieving the Latest Validated Compiler List

```
*   ftp ajpo.sei.cmu.edu
    220 ajpo.sei.cmu.edu FTP server
    (Version 4.98 Wed Feb 19 18:51:48 EST 1986) ready.
*   Name (ajpo.sei.cmu.edu:): anonymous
*   Password (ajpo.sei.cmu.edu:anonymous): your name here
    331 Guest login ok, send ident as password.
    230 Guest login ok, access restrictions apply.
*   ftp> cd public/ada-info
    200 CWD command okay.
*   ftp> mget val-comp.hlp
*   mget val-comp.hlp? y
    200 PORT command okay.
    150 Opening data connection for val-comp.hlp (128.237.2.47,1273)
    (141429 bytes).
    226 Transfer complete.
    143266 bytes received in 6.43 seconds (22 Kbytes/s)
*   ftp> quit
    quit
    221 Goodbye.
```

C.3. Retrieving ASR Files

The Ada Software Repository resides on a machine called SIMTEL20, which is accessible from ARPANET. Unlike the AJPO machine, the operating system is TOPS-20 rather than UNIX. However, the protocol is still the ftp protocol so the scripts are similar. About the only difference is the structure of directory and file names. The following script retrieves a file called "bench.doc" from the "pd2:<ada.benchmarks>" directory. The command "mget PIWG*" could have been used to retrieve all the PIWG benchmarks from the same directory. As for the UNIX machine, the command "cd" will change directories and "dir" will list the contents of the current directory. The master index can be found in the directory "pd2:<ada.master-index>".

```
*   ftp simtel20.army.mil
    Connected to simtel20.army.mil.
    220 WSMR-SIMTEL20.ARMY.MIL FTP Server Process 5Z(53)-7 at
    Fri 4-Nov-88 08:25-MST
*   Name (simtel20.army.mil:nhw): anonymous
*   Password (simtel20.army.mil:anonymous): your name here
    331 ANONYMOUS user ok, send real ident as password.
    230 User ANONYMOUS logged in at Fri 4-Nov-88 08:25-MST, job 20.
*   ftp> cd pd2:<ada.benchmarks>
    331 Default name accepted. Send password to connect to it.
*   ftp> mget bench.doc
*   mget BENCH.DOC.1? y
    200 Port 5.35 at host 128.237.2.47 accepted.
    150 ASCII retrieve of PD2:<ADA.BENCHMARKS>BENCH.DOC.1 started.
    226 Transfer completed. 7291 (8) bytes transferred.
    7291 bytes received in 6.91 seconds (1 Kbytes/s)
*   ftp> quit
    221 QUIT command received. Goodbye.
```

It should be noted that the PIWG tests are also available in the AJPO machine in the directory public/piwg.

Appendix D: Acronyms

AAH	*Ada Adoption Handbook*
ACEC	Ada Compiler Evaluation Capability
ACM	Association for Computing Machinery
ACVC	Ada Compiler Validation Capability
AdaJUG	Ada Joint Users Group
AES	Ada Evaluation System
AFB	air force base
AJPO	Ada Joint Program Office
ANSI	American National Standards Institute
APSE	Ada programming support environment
ARG	Ada Rapporteur Group
ARPA	Advanced Research Projects Agency (now DARPA)
ARTEWG	Ada Runtime Environment Working Group
ASR	Ada Software Repository
AVF	Ada Validation Facility
AVO	Ada Validation Office
AVS	Ada Validation Suite
BSI	British Standards Institute
C^3I	command, control, communications, & intelligence
CACM	*Communications of the ACM*
CD-ROM	Compact Disk Read Only Memory
CECOM	Communications-Electronics Command
CIFO	*Catalog of Interface Features and Options*
COTS	commercial off-the-shelf software
CPU	central processing unit
CRID	*Catalog of Runtime Implementation Dependencies*
DACS	Data Analysis Center for Software
DARPA	Defense Advanced Research Projects Agency
DEC	Digital Equipment Corporation
DoD	Department of Defense
E&V	evaluation and validation
HOL	higher order language
IBM	International Business Machines
ICE	in-circuit emulator
ISA	instruction set architecture
ISO	International Standards Organization
JTC	Joint Technical Committee
MAPSE	minimal Ada programming support environment
MCCR	mission critical computer resources
MDS	microprocessor development system
MIS	management information system
MoD	Ministry of Defense
MVS	Multiple Virtual System
NPL	National Physical Laboratory
PDL	program design language
PIWG	Performance Issues Working Group
REST	Real-Time Embedded Systems Testbed
RM	*Reference Manual*

ROM	read-only memory
SEI	Software Engineering Institute
SIGAda	Special Interest Group on Ada
SQL	structured query language
URG	Uniformity Rapporteur Group
VAX	virtual address extension
VMS	virtual memory system
VSR	validation summary report
WG	working group

Index

ACEC 11, 13, 37, 62, 64, 75, 98
  ACEC Reader's Guide 62
  ACEC User's Guide 98
Acronyms 139
ACVC 17, 52, 69, 131
Ada Adoption Handbook 7
Ada Board 129
Ada Compiler Validation Capability 131
Ada Evaluation System 13, 102, 133
Ada Implementers' Guide 70
Ada Implementors' Guide 52
Ada Information Clearinghouse 129
Ada issues 135
Ada Joint Program Office 129
Ada Joint Users Group 125
Ada Maintenance Organization 130
Ada Programming Support Environment 131
Ada Rapporteur Group 10
Ada Run-Time Environment Working Group 126
Ada Software Repository 13, 107, 136
Ada Validation Facility 19, 130
Ada Validation Office 19
Ada Validation Organization 131
Ada Validation Suite 130
Ada-JOVIAL Newsletter 130
AdaIC 129
AdaJUG 125
Address clauses 46
Aerospace benchmarks 106, 133
AES 13, 41, 75, 102, 133
AJPO 19, 129
AMO 130
ANSI/MIL-STD-1815A 10
APSE 22, 33, 75, 83, 102, 131
ARG 10, 126
ARTEWG 10, 55, 70, 126
ASR 107, 134
ASR Newsletter 134
Asymetrical translation 90
Asynchronous events 89
Attributes 45, 64
  'ADDRESS 64
AVF 19, 130
AVO 19, 131
AVS 17, 130

Bare machine 55
Bare target 19
Base compiler 19
Benchmark
  composite 86
  configuration information 94
  data analysis 92
  strategy 92
  synthetic 88
  timing anomalies 90
  types 85
Benchmarking 22, 85, 126, 131
Boundary alignment 89
British Standards Institute 13, 24, 103, 133

CAIS 131
CALENDAR 90
CECOM 131
Chapter 13 features 11, 18
Checklists 23
Clock 89
Co-processors 70
Code expansion 12, 64
Code inspection 58
Comparing Ada 26
Compile/link time 12, 43
Compiler
  base 19
  capacity 48
  derived 19
  options 18, 43
  performance 46
  project-validated 19
  reevaluation 24
  selection process 29
  tools 75
Composite benchmarks 86
Concurrency 65
Conferences 129
Configurability 69
CRID 70
Cross compiler 55
Cross development
  downloading 79
  environment 32
Cross-development environment 32, 82
Cutter Information Corporation 134
Cycle stealing 89

DACS 98, 133
Daemons 89, 94
Data analysis 92
Debugging 79, 81
Defense Data Network 135
Derived compiler 19, 130
Dhrystone 88
DIANA 77, 83
Documentation 50
DoD Directive 3405.1 1

DoD Directive 3405.2 1
Downloading 79
DTIC 134
Dual loop design 90

E&V Reference Manual 22
E&V team 7, 98, 131
  E&V Guidebook 7
  E&V Reference Manual 7, 22
Efficiency 58
ELABORATE pragma 67
Elaboration 67
Embedded systems 19, 97
Evaluation 22
  benchmarks 22
  completeness 23
  correctness 23
  cost 35
  hardware requirements 14, 31
  information 7
  portability 38
  porting requirements 8
  reevaluation 24
  runtime performance 55
  schedule 35
  software requirements 14, 32
  tailoring 24
  tests 14
  time requirements 7
  usability 23
  vendors 39
Evaluation and Validation Team 131
Evaluation service 24
Exceptions 66
Execution
  storage requirements 68
  time 55, 73

Ftp 135

Garbage collection 69, 90, 105
Generated code
  space efficiency 64
  time efficiency 58
Granularity 85
Grebyn Corporation 134

Hard real-time 15
Hardware clock 74, 89, 91
Host-based compilers 55
Human factors 50

I/O 46
Implementation dependent features 10
Implementation options 52
Implementation-dependent features 11
In-circuit emulator 91
Incremental compilation 76
Input-output packages 66
Inspection 58
Integration 83
International Resource Development, Inc. 134
Interrupt handling 11
Interrupts 72, 89
ISO 126

JTC1 126

Language feature tests 86
Language features 59
Library units 67
Linker 12, 68, 77
Loader 68
Loading, selective 77
Logic analyzer 91

Machine code insertions 46
MAPSE 75, 83
Memory 89
  interleaving 89
  requirements 64
Microprocessor development system 91
MIS 15, 66
Multi-level memories 89

NTIS 134

Optimization 11, 60, 81, 90

Package
  CALENDAR 73, 90
  STANDARD 45
  SYSTEM 45, 73
Performance Issues Working Group 126
Periodic events 89
Pipelined architectures 89
PIWG benchmarks 13, 100, 126
Portability 8, 38
Pragmas 44
Program library system 75
Programming environment 75

Real-time executive 56
Recompilation 76
Reevaluation 24
Rendezvous 65
Representation clauses 46
Runtime 55
Runtime system 89
  space efficiency 68

SC22 126
SIGAda 125
Simtel20 135
Software vernier 90
SQL 126
Stoneman 83
Storage requirements 68
Sublibraries 77
Support tools 12
Synthetic benchmarks 88

Tailoring evaluations 24
Target simulator 82
Tasking 65
Test
  application-specific 88
  capacity 86
  composite benchmarks 86
  degradation 86
  language feature 86
  synthetic benchmarks 88
Test equipment 32
Test hardware 14
Test software 14
Test suite 13, 33
  ACEC 98
  Aerospace Benchmarks 106
  AES 102
  PIWG Benchmarks 100
  requirements 33
  summaries 117
  University of Michigan Ada Benchmarks 105
Testing 59
Text books 129
Time 73
Timing
  anomalies 90
  verification 91
Tools 75
  debugger 79
  integration 83
  program library 75
Training 129

Unchecked conversion 46
Uniformity Rapporteur Group 10, 48, 52
University of Michigan benchmarks 105, 133
URG 10, 48, 52, 71, 126

Validated compiler 19
Validation 10, 17
  ACVC 17
  procedures 17
  project 19
Validation Summary Report 19, 20
Variation 89
Vendors 8
VSR 19, 20

WG9 126
Whetstone 88

Table of Contents

1. Introduction **1**
1.1. Purpose and Scope 1
1.2. Handbook Organization 3
1.3. Tips for Readers 4

2. Common Questions **7**
2.1. Questions About Procedure 7
2.2. Questions About Compiler Technology 10
2.3. Questions About Evaluation Technology 13

3. Compiler Validation and Evaluation **17**
3.1. Validation 17
3.1.1. Validation Procedures 19
3.1.2. Validation Summary Reports 20
3.2. Evaluation 22
3.2.1. Quantitative Criteria and Benchmarks 22
3.2.2. Qualitative Criteria and Checklists 23
3.2.3. Other Evaluation Techniques 24
3.2.4. Reevaluation 24
3.2.5. Tailoring Evaluations 24
3.2.6. Comparing Ada with Other Languages 26

4. Practical Issues of Selecting an Ada Compiler **29**
4.1. Selection Process 29
4.2. Hardware Requirements for Evaluation 31
4.3. Software Requirements for Evaluation 32
4.4. Test Suite Requirements 33
4.5. Timetables, Dependencies, and Costs 35
4.6. Defining Requirements and Criteria 38
4.7. Portability Issues 38
4.8. Evaluating Vendors 39
4.9. Getting More Information 41

5. Compile/Link-Time Issues **43**
5.1. Compiler Options and Special Features 43
5.1.1. Compiler Options 43
5.1.2. Pragmas 44
5.1.3. Attributes 45
5.1.4. Other Important Compiler Features 45
5.2. Compile/Link-Time Performance 46
5.3. Compiler Capacity and Limitations 48

5.4. Human Factors 50
5.4.1. Informational Outputs and Diagnostics 50
5.4.2. Error Recovery 51
5.4.3. Documentation 51
5.5. Implementation Options 52

6. Execution-Time Issues 55
6.1. The Runtime System Model 55
6.2. Time Efficiency of Generated Code 58
6.2.1. Inspection 58
6.2.2. Testing 59
6.2.3. Optimizations Supported 60
6.3. Space Efficiency of Generated Code 64
6.4. Time Efficiency of the Runtime System 65
6.4.1. Tasking 65
6.4.2. Exception Handling 66
6.4.3. Input and Output 66
6.4.4. Elaboration 67
6.5. Space Efficiency of the Runtime System 68
6.6. Features of the Runtime System 69
6.7. Implementation Dependencies 70
6.8. Interrupt Handling 72
6.9. The Clock and Timing Issues 73

7. Support Tool Issues 75
7.1. Program Library System 75
7.1.1. Recompilation and Incremental Compilation Features 76
7.1.2. Sublibraries 77
7.2. Linker/Loader Support 77
7.2.1. Selective Loading 77
7.2.2. Other Linker/Loader Features and Options 78
7.2.3. Downloading for Cross-Development Systems 79
7.3. Support for Debugging 79
7.3.1. Effects of Optimization on Debugging 81
7.3.2. Debugger User Interface 81
7.3.3. Cross Debuggers 82
7.4. Target Simulator 82
7.5. Other APSE Tools 83

8. Benchmarking Issues 85
8.1. Types of Tests 85
8.1.1. Language Feature Tests 86
8.1.2. Capacity and Degradation Tests 86
8.1.3. Composite Benchmarks 86

8.1.4. Synthetic Benchmarks 88
8.1.5. Application-Specific Tests 88
8.2. Factors Causing Variation in Results 89
8.3. Timing Anomalies 90
8.4. Timing Verification 91
8.5. Data Analysis and Reporting 92
8.6. Strategy for Benchmarking 92
8.7. Standard Benchmark Configuration Information 94

9. Test Suites and Other Available Technology 97
9.1. General Information on Evaluation and Test Suites 97
9.2. The Ada Compiler Evaluation Capability 98
9.3. The PIWG Benchmarks 100
9.4. The Ada Evaluation System 102
9.5. The University of Michigan Ada Benchmarks 105
9.6. Aerospace Benchmarks 106
9.7. Ada Software Repository 107
9.8. Other Sources 109

References 111

Appendix A. Test Suite Summaries 117
A.1. ACEC Test Groupings 117
A.2. PIWG Test Groupings 119
A.3. AES Test Groupings 121
A.4. University of Michigan Test Groupings 122
A.5. ACPS Test Groupings 122

Appendix B. Compiler Evaluation Points of Contact 125
B.1. Professional Organizations 125
B.1.1. Ada Joint Users Group (AdaJUG) 125
B.1.2. SIGAda 125
B.1.3. ISO/JTC1/SC22/WG9 126
B.2. U.S. Government Sponsored/Endorsed Organizations 129
B.3. Sources of Evaluation Technology 133
B.4. Ada Information Sources 133

Appendix C. Accessing Network Information 135
C.1. Retrieving Ada Issues 135
C.2. Retrieving the Latest Validated Compiler List 136
C.3. Retrieving ASR Files 136

Appendix D. Acronyms **139**

Index **141**

List of Figures

Figure 4-1: Hypothetical Milestone Chart 36
Figure 6-1: Three Models of Ada Runtime Configuration 57
Figure 6-2: Simple Test for Complex Types 58
Figure 6-3: Simple Test of Procedure 59

List of Tables

Table 3-1: Application Concerns (Hypothetical Example) 25