UNCLASSIFIED

(4)

AR-006-405

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

TECHNICAL REPORT
ERL-0513-TR

ADA COMPILER BENCHMARKING: INITIAL INVESTIGATIONS

by

R. Vernik, I. Turner and S.F. Landherr

SUMMARY

This paper documents some initial investigations into the benchmarking of Ada compilers. A summary of available benchmarking suites is given, although only two of these suites were used in the initial benchmarking experiments: the ACM SIGAda Performance Issues Working Group (PIWG) benchmarks and the University of Michigan (UMICH) benchmarks. Experiences and lessons learned in applying these suites to the Alsys Ada compiler hosted on a Toshiba personal computer and to the DEC VAX Ada compiler hosted on a VAX 8300 are provided. Based on these initial benchmarking experiences, several areas of possible further research/development are identified. In particular, the need for more advanced analysis tools is discussed.

APRIL 1990

COPY No. 11

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1600, Salisbury, South Australia, 5108.

CONDITIONS OF RELEASE AND DISPOSAL

Contents

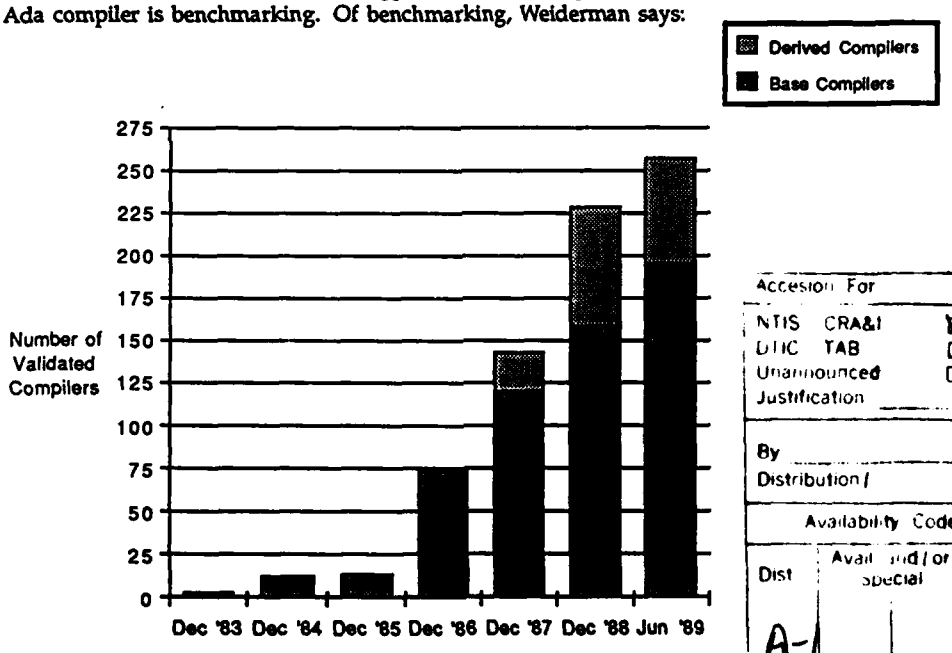
List of Tables

THIS IS A BLANK PAGE

1 Introduction

The number of validated Ada compilers has increased significantly over the past two years, (see Fig 1). This has given the software developer a much wider choice but also brings with it the problem of compiler selection. A common misconception is that because a compiler is validated, it will automatically be a useful development tool for a particular application. Validation simply indicates that the compiler complies with the language standard.[1]. The validation process does not provide information on the quality or characteristics of Ada compilation systems. There may be several compilers which fit the general needs (e.g., host/target combination) but the developer must determine which compilers can support the specific requirements of the application and which compiler will be most effective. This may involve determining which compiler provides the most efficient implementation of Ada tasking, has the lowest subroutine overheads, and provides an effective means of run-time memory management. Moreover, an assessment of compilation speed and library capacity limits may also be important issues in the selection process.

Evaluation is the key to determining whether or not the compiler can be used effectively for software development. There are several aspects that should be considered when selecting a compiler. These are covered in detail by Weiderman in the "Ada Adoption Handbook: Compiler Evaluation and Selection" [2]. As suggested, one technique that can aid in the selection of an Ada compiler is benchmarking. Of benchmarking, Weiderman says:



* *Source : AdaIC Vol VII, No. 1*

Figure 1 Validated Ada Compilers (1983-1989).

"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 can be a very powerful evaluation technique if used properly. However, as with other forms of measurement, care must be taken to understand how to perform the measurements, how to analyze what is being measured, and how these measurements can be used to make valid decisions. Once a commitment has been made to benchmark, the following questions arise: what benchmarking tools are available, where can the tools be obtained, and what problems can be encountered?

Prompted by a number of enquiries regarding Ada compilers, a large number of unanswered questions, and a number of reported potential problems [3][4][5], we decided to gain first-hand experience in this area. The aim of our initial investigation was to:

- provide details of available benchmarking tools and techniques,
- report on our experiences and lessons learned,
- identify areas for future research/development.

To accomplish our aim we based our work on two available benchmarking suites: the University of Michigan (UMICH) benchmarks [6], and the Association of Computing Machinery (ACM) Performance Issues Working Group (PIWG) benchmarks [7]. These were applied to the DEC VAX Ada compiler running on a VAX 8300 and the Alsys Ada compiler running on an IBM compatible Toshiba personal computer (PIWG benchmarks only). This paper reports on these initial experiences in Ada compiler benchmarking and identifies a number of areas for further research/development which will hopefully help make benchmarking a more useful evaluation tool and less of a 'black art'.

2 Available Benchmarking Tools

There are several Ada compiler benchmarking suites currently available. Each is designed to measure certain aspects of the operation and output of an Ada compilation system. The first part of this section discusses the types of tests provided by benchmarking suites. This is followed by an overview of some of the most commonly used suites.

2.1 Types of Tests

The types of tests provided by the benchmarking suites may include tests of:

- **Individual Language Features.** These tests measure characteristics of individual language features such as procedure calls, exception handling, task creation, task rendezvous, and dynamic storage. This information may prove useful if an application is to make heavy use of a particular set of features.
- **Run-time Features.** The characteristics of the run-time system are examined by these tests. This may include examination of memory management and scheduling considerations.
- **Composite Code.** Composite benchmarks test many features in combination. They can take the form of an example application (e.g., a previously developed application which will approximate the proposed development) or smaller known sections of code (e.g., Quicksort, Ackermann's Function).
- **Synthesised Code.** Synthetic benchmarks provide a measure based on some scientifically constructed code. Two of the most widely used synthetic benchmarks for Ada compilers are the Whetstone and Dhrystone. Whetstone is structured towards numerical computation with a heavy emphasis on floating point operations. Dhrystone produces a measure based on what might be expected for typical systems programs using modern programming languages.
- **Code Optimization.** These tests show the effect of optimization on the execution speed and code size.
- **Compilation Times.** These tests provide measurements of the time required for compiling individual features (e.g., incremental time to compile 100 withs on TEXT_IO) or some composite of language features.
- **Library Capacity.** Limits on the size and efficiency of the compiler library system are assessed by these tests.

Organizations wishing to use benchmarks to aid in the selection of an Ada compiler need to determine which measurements are required and then choose the appropriate tools to perform those measurements.

2.2 Benchmarking Suites

Our investigations show that the following Ada specific benchmarking suites are currently in general use:

- **University of Michigan (UMICH) Benchmarks.** The UMICH tests concentrate on measuring individual language features and run-time features. This was one of the earliest Ada benchmark suites and many of its tests and techniques have been incorporated into the more comprehensive PIWG suite. However the UMICH suite may be of value for a more in-depth analysis of some features (e.g., subprogram calls). The problems with this suite are that it is no longer supported and no analysis tools are provided. Documentation is limited to a README file supplied with the suite and a paper by Clapp et al [6].
  References: [6][2][8][9].

- **Performance Issues Working Group (PIWG) Benchmarks.** This suite was prepared by the PIWG of the Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda). The tests have been grouped into three broad categories: composite/synthetic tests, individual timing tests, and compilation tests. The suite is distributed by PIWG and is also available on the Ada Software Repository (ASR) which resides on the SIMTEL20 host computer on the Defence Data Network (DDN). Due to its accessibility, the PIWG suite is widely used by the Ada community. The only documentation supplied is a README file. No analysis tools are provided.
  References: [7][2][8][9].

- **Ada Compiler Evaluation Capability (ACEC).** The ACEC was developed by Boeing Military Aircraft Corporation for the Ada Programming Support Environments (APSE) Evaluation and Validation (E&V) Team of the Ada Joint Program Office (AJPO). The test suite includes: language feature tests, composite and synthetic benchmarks, optimization tests, sorting programs, and example applications. Reported major advantages of the ACEC are that the suite is well documented and there is some automated support for analysis of results. However, a major problem is that the ACEC is currently under U.S. export controls and so may not be readily available to prospective Australian users.
  References: [2][8][9].

- **The Prototype (ACEC) Benchmarks.** This suite was constructed by the Institute for Defence Analyses (IDA) for the E&V Team of the AJPO. It has been superseded by the ACEC. The tests provide timing and storage measurements for individual language features. The suite is available through SofTech Inc. U.S.A.
  References: [4][7][8][9].

- **Benchmark Generator Tool (BGT).** This tool generates benchmarks that measure compiler performance for development machines. Library Capacity Tests and Dependency Maintenance Tests are used to address the problems arising with large system developments. The suite is available on the ASR and is also available through MITRE Corp, McLean, Virginia. The paper by Rainier et al [10] describes the BGT in detail.
  References: [10][8][9].

- **Ada Evaluation System (AES).** The AES was developed for the British government. This suite evaluates Ada compilers and associated linkers/loaders, program library systems, debuggers and run-time libraries. Organizations may purchase a *simplified* version of

the AES (about $US 1,800) or pay the British Standards Institute (about $US 21,600) to carry out a complete evaluation using the Assessor Support System of the AES. Copies of existing reports may also be purchased (about $US 450 for individual reports or about $US 3,600 annually for 12 reports). The major problems are the cost of the suite and its availability in Australia.

References: [2][8][9].

THIS IS A BLANK PAGE

3 Overview of PIWG and UMICH

This section provides a more comprehensive look at the two benchmark suites used for our initial investigations. Appendix IV provides details of the physical make-up of the suites. The PIWG suite covers a wider range of benchmarks than the UMICH suite. In addition to the measurement of individual language and run-time features (the only areas covered by UMICH), PIWG also provides synthetic and composite code measurements, and measurements of compilation/link/execute times. Table 1 gives a summary of the tests provided by the two suites and shows a broad comparison. Comparison is somewhat difficult in some areas because of the different emphasis given to certain features and because of the manner in which the suites are structured. The following paragraphs compare the UMICH and PIWG suites and give an insight into what is actually measured by these suites.

Test	PIWG	UMICH
Chapter 13 Features	Yes	No
CLOCK Resolution and Overhead	Yes	Yes
Coding Style	Yes	No
DELAY Function and Scheduling	Yes	Yes
Dynamic Allocation/Deallocation	Yes	Yes
Exception Handling	Yes	Yes
Loop Overhead	Yes	No
Subprogram Calls	Yes	Yes
Task Creation/Activation	Yes	Yes
Task Rendezvous	Yes	Yes
TEXT_IO Timing	Yes	No
Time Arithmetic	No	Yes
Run-time Memory Management	No	Yes
Composite Benchmarks	Yes	No
Synthetic Benchmarks	Yes	No
Ada Feature Compile Times	Yes	No
Composite Compile/Link/Execute	Yes	No

Table 1 Comparison of PIWG and UMICH.

3.1 Timing Mechanisms

The timing mechanisms used for the PIWG and UMICH benchmarks follow a similar approach. Since both sets of benchmarks were intended for general use, a timing scheme was required which would allow portability of the benchmark software. Ada has a predefined *CLOCK* function which can be used for time measurements. This standard function accesses the underlying system timer to return a time value and so its use in the benchmark programs can help make them system independent. However, in using this approach, the benchmark designers had to overcome a number of problems.

Time resolution was one of the major problems. For example, the simplest way to measure execution time for an individual language feature is to isolate the feature under test and then make time measurements before and after execution. The difference is the time required for the operation. However, to do this, the time resolution of the measurements must be much better than the time required by the operation under test. Time resolution is not specified by the Ada language standard and so there is no guarantee that it will be adequate. For example, the clock resolution for the VAXAda compiler tested is 10 milliseconds, whereas the Alsys compiler can achieve a 1 millisecond resolution. Considering that a procedure call and return may be of the order of 10 microseconds, it is clear some additional techniques must be applied if the Ada *CLOCK* function is to be used.

To overcome these problems a dual loop timing scheme was used. This approach uses a control loop and a test loop (the loops are the same except that the test loop contains the feature to be measured). To obtain the desired resolution, the loops are executed a large number of times. The execution time for the feature under test is computed from the difference in execution times of the two loops. Although simple in concept, there were a number of issues that needed to be considered by the benchmark developers if the benchmarks were to prove useful. These included overcoming the effects of optimizers, ensuring sufficient measurement accuracy, avoiding operating system distortions, and obtaining repeatable results.

Even though these issues were addressed, inaccuracies with dual loop benchmarks have been reported [3][4][11][12]. For example, Donohoe [12] reported that negative values were produced for some of the tests when benchmarking the VAXAda compiler on a MicroVAX II using the UMICH suite. Investigation showed that the VAXELN paging mechanism lengthened the execution of loops that spanned a page boundary. As such, there were cases where the control loop actually took longer to run than the test loop. Clearly, the dual loop approach, although effective for most measurements, can produce inaccurate results and considerable care needs to be taken when interpreting the results.

3.2 Language Feature and Run-time Tests

Both the UMICH and PIWG suites provide tests for the following features:

- **Task Creation and Termination.** PIWG and UMICH provide composite time measurements for task creation and termination. Apparently, individual measurements for elaboration, activation, and termination cannot be provided because of the resolution of the *CLOCK* function [6]. The suites each have three tests covering different scenarios.

- **Exception Handling.** PIWG has five tests to measure the time taken to raise and handle exceptions. Measurements show the effect of exception propagation with different levels

of nesting. Exceptions in task rendezvous are also measured. UMICH also provides a measure of exception propagation delay (although not as comprehensive a range as PIWG). In addition to the tests provided by PIWG, UMICH also provides measurements for various predefined exceptions (e.g., constraint error, numeric error, tasking error).

- **Subprogram Calls.** UMICH provides substantial coverage of subprogram overhead. A good proportion of the UMICH output relates to this area. Times are provided for entering and exiting a subprogram with various scalar parameters and composite objects. The three modes (*IN, OUT, IN OUT*) are covered. There are tests for a wide variety of combinations (e.g., inter-package, intra-package). Measurements are also made for situations where subprograms are part of a *generic* and the effect of using the *INLINE pragma*. The PIWG measurements for subprogram calls are not as comprehensive as UMICH. Even so, there are 11 tests which cover a wide range of possibilities.

- **Dynamic Storage Allocation/Deallocation.** UMICH provides a more comprehensive set of measurements for dynamic allocation/deallocation than PIWG. PIWG has four tests which deal with the allocation and deallocation of a 1000 integer array. UMICH has a considerable number of tests for fixed and variable storage allocation (covering integers, enumeration objects, strings, records, and arrays). Also, there are tests for explicit dynamic allocation using the *new* allocator.

- ***CLOCK* Function Resolution and Overhead.** UMICH and PIWG both provide measurements of *CLOCK* resolution. UMICH also provides a measurement of *CLOCK* function overhead.

- **Task Rendezvous.** PIWG has seven tests for task rendezvous. These tests give rendezvous times for a number of different cases where the number of active tasks, select statements, and entries varies. UMICH provides a single test for task rendezvous.

- ***DELAY* Function and Scheduling.** Both PIWG and UMICH have a test to measure the actual versus requested *delay* for a set of values.

UMICH provides tests for two areas not covered by PIWG:

- **Run-time Memory Management.** The four tests in this section check the memory management characteristics of the run-time system. The test programs use the *new* allocator in a loop to allocate blocks of integers and then provide checks to see whether garbage collection is performed, to find the memory limit, to see if *UNCHECKED_DEALLOCATION* is implemented, and to measure paging times and memory allocation in virtual memory systems.

- **Time Arithmetic.** There are some 15 tests of the arithmetic operators in the standard CALENDAR package. These tests measure the overhead involved in using the "+" and "-" functions of the package. The tests take the form of:

  ```
  Time := Var_Time + Const_Duration
  ```

  where Time is a *TIME* type as returned by the *CLOCK* function and Const_Duration is a *DURATION* value.

The following language feature and run-time tests are provided only by PIWG:

- **Chapter 13 Features.** These PIWG tests provide information on several language features detailed in Chapter 13 of the Ada language reference manual [1]. The tests cover the use of pragma *PACK, UNCHECKED_CONVERSION,* and representation clauses. Nine tests are provided in this area.

- **TEXT_IO Timing.** PIWG provides seven tests covering some of the *TEXT_IO* features. File access measurements are provided using *Get_Line, Put_Line, Get,* and *Put.* Also, there are measurements for reading and writing to local strings using *Put* and *Get.* The final measurement in this group gives the time taken to open and close a file.

- **Loop Overhead.** These tests measure the overhead associated with the *for* loop, the *while* loop, and the use of an *exit* statement within an infinite loop. Two additional tests measure the effect of *pragma OPTIMIZE(TIME),* and *pragma OPTIMIZE(SPACE).*

- **Coding Style.** These tests measure the difference in execution time for coding:

  ```
  Is_Smaller := Number_1 < Number_2;
  ```

  or alternatively doing the same thing by using:

  ```
  if Number_1 < Number_2 then
    Is_Smaller := TRUE;
  else
    Is_Smaller := FALSE;
  end if;
  ```

  This is the only aspect of coding style that is measured.

3.3 Composite/Synthetic Benchmarks

Synthetic benchmarks included in PIWG are:

- **Whetstone Benchmark.** Whetstone provides a single number (Kilo Whetstones per Second) which rates a computer/compiler combination as to how efficiently it executes those features which are most commonly used in actual programs. Originally developed in ALGOL 60, Whetstone reflects numerical computing, particularly floating-point arithmetic.

- **Dhrystone Benchmark.** Dhrystone is similar to Whetstone, however it has been designed to reflect how efficiently systems applications (i.e., applications which place more emphasis on the use of enumeration, record, and pointer data types) will execute. The distribution of Ada features in Dhrystone is based on actual statistics for systems programming applications. Although Dhrystone is more representative of modern programming languages than Whetstone, it does not include features such as tasking or exception handling.

Composite benchmarks provided by PIWG include:

- **Henessy Benchmark.** The Hennessy benchmark is a collection of well-known programming problems such as the Towers of Hanoi, Eight Queens, Quicksort, Bubble Sort, Fast Fourier Transform and Ackermann's function. They can be used for comparing the Ada language with other programming languages.

- **Tracker Algorithm.** There are four tests in PIWG which relate to the "Tracker" application program. The PIWG does not provide information on the rationale for testing this application or how the data can be used. A search of other relevant literature failed to explain its relevance.

3.4 Compilation Time Measurements

The compilation tests in PIWG are in two distinct groups. The first group consists of composite compile/link/execute time measurements for example applications. One of these applications is a program to solve some basic physics problems. It uses a number of packages which must be compiled. PIWG are using the results of these tests to plot industry trends for compiler and environment performance.

The second group (covered in the third run of PIWG) consists of the compile-only tests for various Ada features. These measure things like "the incremental time to compile N nested blocks" and "the incremental time to compile N *withs* on *TEXT_IO*". The tests are sets of increasingly larger compilations which can be used for plots of feature versus compilation time. There are some 71 different measurements made in this run.

THIS IS A BLANK PAGE

4 Experiences and Lessons Learned

One of the major reasons for undertaking this initial study of Ada compiler benchmarking was to gain first-hand experience in the application of the benchmarking suites and to provide details of the lessons learned. This section covers the experience gained and lessons learned by using the PIWG and UMICH benchmarks. Sample outputs have been provided in the appendices for both the VAX/UMICH and VAX/PIWG combinations. Appendix I contains portions of runs 1, 2, and 3 for the VAX/PIWG combination and the outputs of c_run, i_run, and tm_run for the VAX/UMICH combination are provided in Appendix II. Hopefully, the information in this section will aid others who are (or are considering) benchmarking Ada compilers.

Table 2 shows the computer/compiler combinations that were used to run the two suites.

Some of the lessons learned from our initial experiences in Ada compiler benchmarking include:

- The lack of documentation and support can make benchmarking a difficult and time consuming task.
- Compiling, linking, and running the benchmarking suites requires a considerable amount of computer resources (CPU time and secondary storage).
- The lack of analysis tools undermines the ability to obtain clear and concise results from the output generated by a given suite.
- The results obtained may lack accuracy and may be erroneous.
- Considerable expertise and experience is needed if benchmarking is to be successful.
- The appropriate suite needs to be selected in order to obtain the required information.

4.1 Lack of Documentation and Support

A major problem with both the UMICH and PIWG suites is that they both lack documentation and support. README files are supplied with both suites but the information they provide is not comprehensive. In trying to get the chosen suite to run to completion it is likely that unexpected problems will be encountered for which no documentation exists. As with many areas in the

Benchmark Suite	Compiler	Op Sys	Host	Target
UMICH	VAX Ada V1.5	VAX/VMS 4.7	VAX 8300	VAX 8300
PIWG	VAX Ada V1.5	VAX/VMS 4.7	VAX 8300	VAX 8300
PIWG	Alsys V3.2	MS-DOS 3.3	Toshiba 5100	Toshiba 5100

Table 2 Computer/Compiler combinations.

computing field, without documentation even the simplest problems can take hours or even days to solve. For example, some of the problems that we encountered included:

- **"Insufficient Virtual Memory" errors were raised with the VAX/PIWG combination.** During the third run (compile time measurements) of the PIWG suite, INSVIRMEM errors were raised (specifically, while trying to compile the tests Z000172 and Z000173, which measure the incremental time to instantiate 200 and 500 integer_io(integer) packages respectively). The problem was solved by increasing the amount of virtual memory available to the compilation processes. This was achieved by a 'trial and error' adjustment of certain system parameters (*Working Set Extent* and *Page File Quota*). Varying the load on the system together with adjustment of vital system parameters will affect the timing measurements of the tests. These effects need to be understood and taken into account. Tests need to be re-run after adjustments are made.

- **Storage errors were encountered with the Alsys/PIWG combination.** These errors were raised by the compiler during the first and third runs of the suite (execution performance measurements and compile time measurements respectively). Adjustment of the number of buffers used for internal data structures (using the *MAC_BUFFERS* option of the *COMPILE* command) and the size of the heap eliminated these errors in the first run but they were still present during the third run at the time this paper was written and are still under investigation. The problem with adjusting the buffers is that it will have an effect on the timing measurements being taken.

- **A File Creation Error was raised by the operating system with the Alsys/PIWG combination.** MS-DOS reported a *File Creation Error*because DOS was not allowing a sufficient number of open files to access system calls. Adjustment of the *FILES* variable in *CONFIG.SYS* solved the problem.

4.2 Usage of Computer Resources

Running the suites consumed a considerable amount of computer resources; this needs to be considered and planned for (e.g., the VAX/PIWG (third run) took approximately 2 hours of CPU time and 4 hours real time to complete). Running the suites on a non-dedicated time-shared system will have a notable effect on the response times of other processes as well as distorting the measurements that are being made by the benchmarking processes. If benchmarking is to be used to aid compiler selection, then hardware must be dedicated to the task. A measurement plan needs to be developed so that benchmark results can be obtained for different loading levels. Personnel performing the benchmarks will need to be able to control these loading levels so that meaningful results can be derived. In our case, the only time that we could gain "dedicated" access to the VAX computer was after hours.

4.3 Lack of Analysis Tools

No analysis tools are supplied with either the UMICH or PIWG suites, which makes analysis of the considerable amount of data that is produced an involved and lengthy process. In the case of the PIWG suite, the Performance Issues Working Group *itself* carries out analysis of the data that is sent back to them from organizations that have run the suite on their specific computer/compiler combination(s). PIWG expect to receive back the best repeatable time (BRT) for each of its tests. The TAPEDIST.LTR file supplied with the suite states:

"PLEASE send at least one measurement. If you can, make a second and third run to determine stability. If more than one run is made, supply the average and the number of runs averaged. Throw out anomalous large and small runs. We want the best repeatable time that can be achieved without changing the test suite."

A portion of the output produced by the first run of the PIWG suite is shown in Appendix I. Having to read through several such output files in order to determine the BRT for each test is both tedious and time consuming.

As part of this initial study, a program was written in Ada to help determine, from the output produced by the PIWG suite, the BRT for each test. The program's input consists of a number of output files produced by running the PIWG suite. Each test's CPU time is read from each of the files, forming a list of times for each test. The list is then sorted into ascending order and the BRT extracted (if one exists) by comparing each value with the next value in the list. If two of the values are within five percent of each other then the lower is reported as being the BRT. Appendix III contains the output produced by the analysis program.

The analysis program output displays the necessary information in a much more concise format and also makes the detection of unusual results (e.g., zero or negative results) much easier. This program is simple and did not take long to develop but the inclusion of such tools in benchmark suites would make the task of the *benchmarker* much faster and simpler.

4.4 Accuracy of the Results

The measurements obtained from a given suite are dependent on factors such as the architecture of the system, the system software, other applications that are present on the system, and the construction of the tests themselves. These factors may cause unacceptable results (e.g., negative, non-repeatable positive or zero measurements) that need to be detected and explained. Both negative and zero results were produced during the initial investigations described here (negative and zero results from the VAX/UMICH runs and zero results from the PIWG combinations). Much work has been done by the Software Engineering Institute (SEI) dealing with timing issues of Ada benchmarks. Weiderman in [2] lists the factors which need to be considered:

- **Memory effects:** Cycle stealing, Boundary alignment, Memory interleaving, Multi-level memories.
- **Processor effects:** Pipelined architectures, Interrupts, Clocks.
- **Operating and run-time system effects:** General overhead, Periodic and asynchronous events, Garbage collection, Multiprogramming.
- **Program translation effects:** Optimization, Asymmetrical translation, Hidden parallelism.

all of which are explained in detail in [5][11][2].

The point being made is that simply running benchmarks then making decisions based on the results produced could prove to be misleading. Even seemingly acceptable results should be treated with caution.

4.5 Learning Curve

As with any other area, benchmarking has an associated learning curve. The potential benchmarker will need to:

- **Have a sound knowledge of the Ada language.** Although the UMICH and PIWG suites could be *run* without any knowledge of the Ada language itself, *understanding* the results requires a good knowledge of Ada. If erroneous results are encountered then understanding why they occurred would most likely involve delving into the source code of the tests and the timing mechanisms provided.

- **Know how to use the compiler(s) of interest.** The environments provided with their respective compilers are likely to be quite different and a person intending to benchmark a given number of compilers would need to become proficient at using each of them.

- **Have a sound knowledge of the operating environments being used.** If the compiler/benchmark suite combinations are to be run under different operating systems then the benchmarker will need to be familiar with each of them. The task of creating directories to house the files associated with a given suite will need to be carried out (e.g., the source files, the output files, Ada libraries, etc., are likely to be kept in their own directories). The command files/scripts that are used to compile/link/run the tests may need to be modified, which requires a knowledge of the operating system's command language. Knowledge of the system software will also be necessary when analyzing the produced results (e.g., the actions of system processes will need to known and understood).

- **Have a sound knowledge of the chosen machine's architecture.** Knowledge of the peripheral devices, memory, interrupts, clocks, etc., will be needed if the results are to be understood.

- **Know about benchmarking techniques and pitfalls.** The techniques used to construct the tests and the pitfalls involved with this process should be known and understood by the benchmarker. If the performance of a number of compilers is being measured to aid selection then it is essential to ensure that the tests being used accurately reflect the actual workload that the compiler will be placed under. Comparison between results can lead to inaccurate conclusions (e.g., if a test's code needs to be changed to allow it to be compiled by a number of compilers, what are the effects of the changes). If the compilers are being tested on different machines, how comparable are those machines (such things as memory size, need to be taken into account). See [13] for a discussion of these topics.

In short, the solutions to the problems that may arise will involve a sound knowledge of several different areas. If the benchmarker does not possess this knowledge then time and effort will be spent gaining it and the process of benchmarking could turn out to be both costly and time consuming.

4.6 Suite Selection

Before an appropriate benchmarking suite can be selected, the required measurements must be identified (e.g., compilation time, link time, execution efficiency of the generated code, library capacity, etc.). If benchmarking is being used as an aid in selecting a compiler for a specific project, the project's software requirements would need to be well defined and understood so that they

can be correlated with compiler measurements. The benchmarker may find that a single suite will not cover all that is required; supplementing part or all of one suite with part or all of another or constructing custom benchmarks may be the only way to obtain the required information.

THIS IS A BLANK PAGE

5 Future Development/Research Areas

Our initial Ada compiler benchmarking experiences have highlighted a number of areas where additional research/development is needed. These include:

- evaluating other benchmark suites and techniques,
- gaining experience in benchmarking cross compilers,
- defining tools for analysis of results,
- experimenting with the use of hybrid measurement techniques for benchmarking.

5.1 Other Suites and Techniques

The extent of our benchmarking experience is limited to the application of the PIWG and UMICH benchmarks. Although these suites can provide some valuable information on Ada compilers, we discovered several areas where they were limited. These include:

- inadequate documentation,
- lack of analysis tools,
- lack of measurements for code size,
- no assessment of compiler limitations for large developments,
- no assessment of performance for a complete application.

The recently released ACEC is reported to have overcome some of these deficiencies [12]. This suite is far more extensive than either the PIWG or UMICH, is well documented, and could well form the baseline for Ada compiler benchmarking. As such, the ACEC warrants further investigation. A major problem for Australian organizations wishing to use these benchmarks is that the ACEC is controlled by U.S. export restrictions. If the ACEC cannot be obtained because of these restrictions, the Australian Ada community will need to look closely at how the available suites can be consolidated and enhanced to provide a comprehensive and usable set of tools.

A significant risk area in Ada development is the ability of the Ada compilation system to handle large quantities of Ada code. Failure to determine the compiler's characteristics in this area could lead to a 'midstream' change of compilers. This could result in time and cost overruns because of the time taken to select the new compiler and to integrate the new compiler into the development environment, loss of development continuity, and retraining. The BGT has been developed to help prevent these problems by uncovering limits in Ada compilation systems during the Ada compiler selection process. Since the Australian defence industry will soon be engaged in some large scale Ada developments, use of the BGT should be considered and so warrants further investigation.

Experience in the application of several benchmark suites would allow the benchmarks to be categorized as to their applicability, ease of use, accuracy, availability, and support. In addition,

based on this experience, a method for the effective application of benchmarks and the analysis of results could be defined to help formalize the benchmarking process.

Techniques other than those used by the currently available benchmark suites also warrant further investigation. The performance of a complete application cannot be predicted purely by the measurement of individual language features. For example, Ada rendezvous times may be acceptable when measured in isolation, but what is the effect if a number of tasks are run concurrently? Benchmarks such as PIWG give some useful information for comparisons of Ada compilers, but do not address these loading effects. One way to overcome this problem is to use a representative system for compiler evaluation (one that exhibits the same characteristics as the proposed system). Another approach is to use a technique such as that used in the Benchmark Synthesis System [14]. Here, the basic concept is to use a mechanism for describing the anticipated load (a load description language), synthesize Ada code from this scale model, then execute the instrumented Ada code in the target environment. Problems to be addressed for this approach would include the effects of compiler optimization, and clock resolution for timing purposes.

5.2 Benchmarking Cross Compilers

As Ada becomes more widely used in time-critical applications and embedded systems, more emphasis will be placed on benchmarking cross compilers. The problems of benchmarking host compilers are reasonably well understood and recorded [5]. However, a whole set of additional problems and experiences is associated with benchmarking cross compilers [3]. There are a number of questions which need to be answered. For example, are the current benchmarks sufficiently accurate to perform fine-grain analysis for time-critical applications? What is the best way to benchmark a cross-compiler? What additional tools and techniques are required to effectively perform such benchmarks?

5.3 Analysis Tools

As mentioned earlier, a major problem with the PIWG and UMICH benchmarks is that they lack analysis and data reduction tools. Since the suites include some 136 and 150 different tests respectively, analysis tools are essential if effective use is to be made of the results. The situation could be even worse for the more advanced suites such as the ACEC (which includes over 1000 tests) if sufficient analysis support is not provided. Tools need to be provided to analyze the vast amounts of data provided by the benchmark suites. Some areas where analysis tools could be used include:

- comparison of features for different implementations,
- identification of inconsistent results,
- repeatability analysis,
- aids for interpreting results.

The ACEC is reported to include a tool which performs statistical analysis of the results collected from several target systems. Although this is a start, the need for further automation in the analysis subsystem has been reported [2]. An area of further research may be to investigate the tools necessary for the analysis and reporting of benchmark results, and to define how these

tools could be integrated into an overall benchmark analysis and reporting environment. The use of graphical techniques for the comparison of results should form part of the investigation. Perhaps the ACEC (if available) could be used to perform measurements for such an environment.

5.4 Hybrid Measurement Techniques

There are cases where the measurement of individual language features using benchmarking suites such as PIWG and UMICH provides erroneous results. Indeed, the problems associated with the lack of precision of the Ada clock and dual loop benchmarks are well documented [3][11][2]. Negative or zero results can be expected because of these problems. Additional techniques need to be employed to study these erroneous results, provide for fine-grain analysis of language features, and to verify timing results.

Hardware measurement tools have been used to verify benchmark timing results. For example, the SEI used a Gould K115 logic analyzer to measure task rendezvous times for the Systems Designers (SD) Ada-Plus MC68020 cross compiler [3]. This involved examining assembly code and load maps, allowing for word boundaries, calculating offsets, and instrumenting the hardware. This complex process limited the usefulness of the tool.

To overcome some of the problems of directly using a hardware measurement tool, a hybrid measurement approach may be possible. This would involve using optimized software probes, a piece of purpose-built hardware attached to the computer bus, and a general purpose measurement device such as a logic analyzer. The software probes would trigger the purpose-built hardware (perhaps external registers) and the logic analyzer would make measurements on this external hardware. The probes would be inserted at the source code level at the feature to be measured and so the tedious and error-prone process of examining assembler code and load maps for each measurement would be eliminated. This technique has been used successfully to measure performance of computer systems [15] and warrants further investigation for use in benchmarking.

THIS IS A BLANK PAGE

6 Conclusion

Benchmarking can be a very useful technique for evaluating Ada compilers. We found that in addition to obtaining a quantitative assessment, benchmarking also provided a qualitative assessment of the compilers and operating environment. By running the PIWG and UMICH suites, we found that we gained considerable insight into its ease of use, reliability, and integration with other tools and operating environments.

Our initial experiences have highlighted a number of potential problems. One of the major problems is the skill level and experience required to perform the benchmarks and analyze the results. This is definitely not the domain of the novice programmer. Personnel involved in benchmarking should have a sound systems background, understand the limitations of the tools, and understand the principles of measurement. Management should be aware that there needs to be some investment in time and resources if benchmarking is to be undertaken. The results provided by a 'half-hearted' approach to benchmarking could lead to poor decisions which may translate to increased project risk and cost.

Ada compiler benchmarks are available to make a variety of measurements. Benchmarks can provide data on individual language features, compiler limitations, compiler performance, and loading effects. Clearly, if benchmarks are going to be used to help evaluate and select an Ada compiler, a measurement plan needs to be defined, outlining what is to be measured, which tools are to be used, and how the measurements are to be analyzed.

There are several areas where additional research and development is needed to help support the benchmarking process. One of the major areas is the use of analysis tools. The large amounts of data provided by benchmark suites need to be processed into a more readable form. This would then help facilitate compiler comparisons, help identify erroneous or inaccurate measurements, and in general aid the compiler selection process. Additional analysis tools need to be developed and the use of graphics for displaying results should be considered. Finally, hybrid measurement techniques show promise for validating measurements provided by the benchmark suites and for providing 'fine-grain' analysis of language features. This technique warrants further investigation.

THIS IS A BLANK PAGE

References

[1] Department of Defence, Washington, D.C., *Ada Programming Language*. ANSI/MIL-STD-1815A-1983.

[2] Weiderman, N., "Ada Adoption Handbook", Tech. Rep. CMU/SEI-87-TR-13, Software Engineering Institute, March 1989.

[3] Donohoe, P., "Ada Performance Benchmarks on the MC68020: Summary and Results, Version 1.0", Tech. Rep. CMU/SEI-87-TR-40, Software Engineering Institute, December 1987.

[4] Landherr, S., et al., "Evaluation of Ada Environments (chapter 8) ACEC", Tech. Rep. CMU/SEI-87-TR-1, Software Engineering Institute, March 1987.

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

[6] Clapp, Russell M., et al., "Towards Real-Time Performance Benchmarks for Ada", *Communications of the ACM*, vol. Vol. 29, pp. 760–778, August 1986.

[7] Donohoe, P., "A Survey of Real-Time Performance Benchmarks for the Ada Programming Language", Tech. Rep. CMU/SEI-87-TR-28, Software Engineering Institute, December 1987.

[8] Air Force Wright Aeronautical Laboratories, Wright-Patterson Air Force Base, OH, *E&V Guidebook, Version 1.1*, August 1988. AFWAL TR-5234-4.

[9] Air Force Wright Aeronautical Laboratories, Wright-Patterson Air Force Base, OH, *E&V Reference Manual, Version 1.1*, October 1988. AFWAL TR-5234-3.

[10] Rainier, Stephen R., et al., "The Benchmark Generator Tool: Measuring Ada Compilation System Performance", in *Third International IEEE Conference on Ada Applications and Environments*, 1988.

[11] Altman, N. and Weiderman, N., "Timing Variation in Dual Loop Benchmarks", Tech. Rep. CMU/SEI-87-TR-21, Software Engineering Institute, October 1987.

[12] Donohoe, P., "Ada Performance Benchmarks on MicroVAX II: Summary and Results, Version 1.0", Tech. Rep. CMU/SEI-87-TR-27, Software Engineering Institute, December 1987.

[13] Dongarra, J., et al., "Computer benchmarking: paths and pitfalls", *IEEE Spectrum*, pp. 38–43, July 1986.

[14] Knight, John C., and Crowe, Richard H., "A System for Evaluating Ada Implementations Using Synthesized Benchmarks", in *Third International IEEE Conference on Ada Applications and Environments*, 1988.

[15] McKerrow, P., *Performance Measurement of Computer Systems*. Addison-Wesley, 1988.

[16] Hook, A. A., et al., *User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1*. Institute for Defence Analyses, October 1985.

THIS IS A BLANK PAGE

Appendix I.
Portions of the VAX/PIWG Output.

Portion of the output of RUN 1 (produced by COMPILE.COM).

```
Test Name:   A000090
 Clock resolution measurement running
Test Description:
 Determine clock resolution using second differences
 of values returned by the function CPU_Time_Clock.

 Number of sample values is       12000
 Clock Resolution           =        0.009948730468750 seconds.
 Clock Resolution (average) =        0.009948730468750 seconds.
 Clock Resolution (variance) =       0.000000000000000 seconds.


Test Name:   A000091            Class Name:  Composite
           0.9250 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark

Test Name:  A000093                  Class: Composite

Average time per cycle :   786.79 milliseconds

Average Whetstone rating :  1271 KWIPS

Test Description:
 ADA Whetstone benchmark using standard internal math routines


Test Name: A000094                   Class: Composite
   Perm    2.09
 Towers    3.66
 Queens    1.17
  Intmm    1.09
     Mm    1.08
 Puzzle    9.05
  Quick    1.03
 Bubble    1.58
   Tree    1.79
    FFT    1.90
    Ack   71.53
```

Test Description:
Henessy benchmarks

B000001 application program, tracker
TRACK USING COVARIANCE MATRIX

Time Required : 6.71900E+01 Seconds for 10000 Repetitions

TRACK USING COVARIANCE MATRIX - SUPPRESS

Time Required : 4.72600E+01 Seconds for 10000 Repetitions

B000002 application program, tracker
TRACK WITH COVARIANCE MATRIX FLOAT 6 DIGITS

Time Required : 2.83100E+01 Seconds for 10000 Repetitions

TRACK WITH COVARIANCE MATRIX FLOAT 6 DIGITS - SUPPRESS

Time Required : 1.91500E+01 Seconds for 10000 Repetitions

B000003 application program, tracker
TRACK WITH COVARIANCE MATRIX - FLOAT 9 DIGITS

Time Required : 4.46700E+01 Seconds for 10000 Repetitions

TRACK WITH COVARIANCE MATRIX - FLOAT 9 DIGITS SUPPRESS

Time Required : 3.41500E+01 Seconds for 10000 Repetitions

B000004 application program, tracker
TRACK WITH COVARIANCE MATRIX - FLOAT INTEGER

Time Required : 3.38700E+01 Seconds for 10000 Repetitions

TRACK WITH COVARIANCE MATRIX - FLOAT INTEGER SUPPRESS

Time Required : 1.95000E+01 Seconds for 100000 Repetitions

Test Name: C000001 Class Name: Tasking
CPU Time: 7600.1 microseconds
Wall Time: 8200.1 microseconds. Iteration Count: 2
Test Description:
Task create and terminate measurement
with one task, no entries, when task is in a procedure

```
 using a task type in a package, no select statement, no loop,


Test Name:   C000002                          Class Name:  Tasking
CPU Time:     7450.0  microseconds
Wall Time:    7450.0  microseconds.           Iteration Count:   2
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop


Test Name:   C000003                          Class Name:  Tasking
CPU Time:     7600.4  microseconds
Wall Time:    7549.7  microseconds.           Iteration Count:   2
Test Description:
 Task create and terminate time measurement
 Task is in declare block of main procedure
 one task, no entries, task is in the loop
        .       .       .       .       .       .
        .       .       .       .       .       .
```

Portion of the output of RUN 2 (produced by ZCOMPILE.COM).

```
$ RUN A000051
 CPU time now=     14.5600         WALL time now=  42614.1800 seconds.
$ RUN A000051 ! calibrate time to measure time
 CPU time now=     15.0900         WALL time now=  42616.6400 seconds.
$ RUN A000051
 CPU time now=     15.5600         WALL time now=  42620.4800 seconds.
$ ADA   Z000001 ! FLTIO
$ ADA   Z000002 ! REFUNCT
$ ADA   Z000003 ! PREAL
$ ADA   Z000004 ! PUBASIC
$ ADA   Z000005 ! PUMECH
$ ADA   Z000006 ! PUELEC
$ ADA   Z000007 ! PUOTHER
$ ADA   Z000008 ! MKSPMECH
$ ADA   Z000009 ! MKSPELEC
$ ADA   Z000010 ! PCONSTANT
$ ADA   Z000011 ! PUOBASIC
$ ADA   Z000012 ! PUOMECH
$ ADA   Z000013 ! PUOELEC
$ ADA   Z000014 ! PCCONST
```

```
$ ADA   Z000015 ! PUCONV
$ ADA   Z000016 ! PUCMKS spec
$ ADA   Z000016A ! PUCMKS body
$ ADA   Z000017 ! PUCENGL spec
$ ADA   Z000017A ! PUCENGL body
$ ADA   Z000018 ! PHYSICS1
$ ACS LINK Z000018
$ RUN Z000018
 Test printout and value of acceleration,
 9.80665E+00 meter per second squared  = G
 1.10325E+01 meter
 1.50000E+00 second
 2.08030E+01 meter per second
$ ADA   Z000020 ! GENPREAL
$ ADA   Z000021 ! ALLSTMT
$ ADA   Z000022 ! GENSORTSH
$ ADA   Z000023 ! GENSHELLI
$ ACS LINK Z000023
$ RUN Z000023
 UP SORTED DATA
          1  1.00000E+00 AAA FIRST   1.09
          2  2.00000E+00 BBB SECOND   2.09
          3  3.00000E+00 CCC THIRD   3.09
          4  4.00000E+00 DDD FOURTH   4.09
 DOWN SORTED DATA
          4  4.00000E+00 DDD FOURTH   4.09
          3  3.00000E+00 CCC THIRD   3.09
          2  2.00000E+00 BBB SECOND   2.09
          1  1.00000E+00 AAA FIRST   1.09
 in the bag
 gone fishing
 end FISH
 ALL_STATEMENTS_PROCEDURE_2
 into LOOP_NAME_1
 Z000021 finished
```

Portion of the output of RUN 3 (produced by Z00011D.COM).

```
$ ADA Z000111 ! just invoke compiler to get some memory
$ RUN A000052 ! the executable comes from the SECOND RUN
$ RUN A000053
$ RUN A000054
$ ADA Z000110
$ RUN A000055 ! time for minimum compile    (1)
Measurement
CPU Time:        2.85  seconds
```

```
Wall Time:       6.12  seconds
$ RUN A000052 ! check that  (3) about (2) - (1)
$ RUN A000053
$ RUN A000054
$ ADA Z000111
$ RUN A000055 ! time to compile 100 INTEGER declarations (2)
Measurement
CPU Time:        7.25  seconds
Wall Time:      12.46  seconds
$ RUN A000052
$ ADA Z000110
$ RUN A000053
$ RUN A000054
$ ADA Z000111
$ RUN A000055 ! incremental time to compile 100 INTEGER declarations (3)
Measurement
CPU Time:        4.03  seconds
Wall Time:       6.12  seconds
$ RUN A000054
$ ADA Z000111
$ RUN A000055 ! incremental time to compile 100 INTEGER declarations
Measurement
CPU Time:        4.22  seconds
Wall Time:       6.55  seconds
$ RUN A000054 ! check against previous for consistancy
$ ADA Z000112
$ RUN A000055 ! incremental time to compile 200 INTEGER declarations
Measurement
CPU Time:        7.60  seconds
Wall Time:       9.69  seconds
$ RUN A000054
$ ADA Z000113
$ RUN A000055 ! incremental time to compile 500 INTEGER declarations
Measurement
CPU Time:       18.51  seconds
Wall Time:      29.02  seconds
$ RUN A000054
$ ADA Z000114
$ RUN A000055 ! incremental time to compile 1000 INTEGER declarations
Measurement
CPU Time:       36.97  seconds
Wall Time:      43.07  seconds
$ RUN A000054
$ ADA Z000121
$ RUN A000055 ! incremental time to compile and initialize 100 INTEGERS
Measurement
CPU Time:        6.07  seconds
Wall Time:       8.07  seconds
```

```
$ RUN A000054
$ ADA Z000122
$ RUN A000055 ! incremental time to compile and initialize 200 INTEGERS
Measurement
CPU Time:       11.89  seconds
Wall Time:      14.99  seconds
$ RUN A000054
$ ADA Z000123
$ RUN A000055 ! incremental time to compile and initialize 500 INTEGERS
Measurement
CPU Time:       29.38  seconds
Wall Time:      33.93  seconds
$ RUN A000054
$ ADA Z000124
$ RUN A000055 ! incremental time to compile and initialize 1000 INTEGERS
Measurement
CPU Time:       58.42  seconds
Wall Time:      61.70  seconds
$ RUN A000054
$ ADA Z000131
$ RUN A000055 ! increnemtal time to compile and init 100 INTEGER array
Measurement
CPU Time:        3.60  seconds
Wall Time:       7.64  seconds
$ RUN A000054 ! each component named, in reverse order
$ ADA Z000132
$ RUN A000055 ! increnemtal time to compile and init 200 INTEGER array
Measurement
CPU Time:        5.46  seconds
Wall Time:       5.49  seconds
$ RUN A000054 ! each component named, in reverse order
        .       .       .       .       .       .
        .       .       .       .       .       .
```

Appendix II.
Portions of the VAX/UMICH Output.

c_run.log

Subprogram Overhead (generic, cross package)
Number of Iterations = 10000 * 10

Time (microsec.)	Direction Passed	# Passed in Call	Type Passed	Size of Passed Var
16.0		0		
22.2	I	1	INTEGER	
25.0	O	1	INTEGER	
27.6	I_O	1	INTEGER	
26.5	I	10	INTEGER	
50.5	O	10	INTEGER	
83.7	I_O	10	INTEGER	
329.2	I	100	INTEGER	
856.8	O	100	INTEGER	
1324.0	I_O	100	INTEGER	
22.7		1	ENUMERATION	
27.9	O	1	ENUMERATION	
28.7	I_O	1	ENUMERATION	
43.4	I	10	ENUMERATION	
87.7	O	10	ENUMERATION	
99.6	I_O	10	ENUMERATION	
432.5	I	100	ENUMERATION	
472.2	O	100	ENUMERATION	
947.2	I_O	100	ENUMERATION	
19.4	I	1	ARRAY of INTEGER	1
17.5	O	1	ARRAY of INTEGER	1
19.8	I_O	1	ARRAY of INTEGER	1
18.4	I	1	ARRAY of INTEGER	10
27.4	O	1	ARRAY of INTEGER	10
15.5	I_O	1	ARRAY of INTEGER	10
24.0	I	1	ARRAY of INTEGER	100
17.5	O	1	ARRAY of INTEGER	100
37.1	I_O	1	ARRAY of INTEGER	100
19.0	I	1	ARRAY of INTEGER	10000
21.4	O	1	ARRAY of INTEGER	10000
17.2	I_O	1	ARRAY of INTEGER	10000
19.4	I	1	RECORD of INTEGER	1
20.0	O	1	RECORD of INTEGER	1
28.3	I_O	1	RECORD of INTEGER	1
18.4	I	1	RECORD of INTEGER	100
20.1	O	1	RECORD of INTEGER	100

```
|   26.9   |   I_O    |       1 |RECORD of INTEGER    |      100  |
```

i_run.log

```
                        Subprogram Overhead (inline)
Number of Iterations = 10000 * 10

   Time     |Direction|# Passed|         Type          |  Size of  |
(microsec.)|  Passed |in Call |        Passed         |Passed Var|

|Raw Time for TEST #1:     81.7 Raw Time for CONTROL #1:     81.3
|Raw Time for TEST #2:     87.5 Raw Time for CONTROL #2:     90.9
|Raw Time for TEST #3:     84.6 Raw Time for CONTROL #3:     86.0
|     0.4   |          |       0 |                       |          |
|Raw Time for TEST #1:    148.6 Raw Time for CONTROL #1:    137.4
|Raw Time for TEST #2:    128.2 Raw Time for CONTROL #2:    147.4
|Raw Time for TEST #3:    149.9 Raw Time for CONTROL #3:    138.6
|   -9.2    |   I      |       1 |INTEGER                |          |
|Raw Time for TEST #1:    150.7 Raw Time for CONTROL #1:    158.6
|Raw Time for TEST #2:    132.3 Raw Time for CONTROL #2:    146.5
|Raw Time for TEST #3:    145.5 Raw Time for CONTROL #3:    150.3
|  -14.2    |   O      |       1 |INTEGER                |          |
|Raw Time for TEST #1:    119.3 Raw Time for CONTROL #1:    128.9
|Raw Time for TEST #2:    105.3 Raw Time for CONTROL #2:    111.1
|Raw Time for TEST #3:    138.8 Raw Time for CONTROL #3:    130.8
|   -5.8    |   I_O    |       1 |INTEGER                |          |
|Raw Time for TEST #1:    104.6 Raw Time for CONTROL #1:    111.9
|Raw Time for TEST #2:     91.3 Raw Time for CONTROL #2:     97.7
|Raw Time for TEST #3:     87.4 Raw Time for CONTROL #3:     93.1
|   -5.7    |   I      |      10 |INTEGER                |          |
|Raw Time for TEST #1:    139.8 Raw Time for CONTROL #1:    136.8
|Raw Time for TEST #2:    129.0 Raw Time for CONTROL #2:    125.8
|Raw Time for TEST #3:    117.3 Raw Time for CONTROL #3:    129.4
|   -8.5    |   O      |      10 |INTEGER                |          |
|Raw Time for TEST #1:    189.6 Raw Time for CONTROL #1:    179.8
|Raw Time for TEST #2:    184.5 Raw Time for CONTROL #2:    165.2
|Raw Time for TEST #3:    176.4 Raw Time for CONTROL #3:    166.6
|   11.2    |   I_O    |      10 |INTEGER                |          |
|Raw Time for TEST #1:    680.8 Raw Time for CONTROL #1:    341.7
|Raw Time for TEST #2:    729.7 Raw Time for CONTROL #2:    368.4
|Raw Time for TEST #3:    697.2 Raw Time for CONTROL #3:    369.6
|  339.1    |   I      |     100 |INTEGER                |          |
|Raw Time for TEST #1:   1123.6 Raw Time for CONTROL #1:    610.4
|Raw Time for TEST #2:   1032.5 Raw Time for CONTROL #2:    550.8
|Raw Time for TEST #3:   1124.5 Raw Time for CONTROL #3:    674.3
|  481.7    |   O      |     100 |INTEGER                |          |
|Raw Time for TEST #1:   2008.0 Raw Time for CONTROL #1:   1668.5
|Raw Time for TEST #2:   2024.4 Raw Time for CONTROL #2:   1785.1
|Raw Time for TEST #3:   2149.3 Raw Time for CONTROL #3:   1957.3
```

339.5	I_O	100	INTEGER				
Raw Time for TEST #1:		130.4	Raw Time for CONTROL #1:	149.5			
Raw Time for TEST #2:		152.0	Raw Time for CONTROL #2:	117.9			
Raw Time for TEST #3:		133.7	Raw Time for CONTROL #3:	150.1			
12.5	I	1	ENUMERATION				
Raw Time for TEST #1:		116.7	Raw Time for CONTROL #1:	158.2			
Raw Time for TEST #2:		148.6	Raw Time for CONTROL #2:	130.5			
Raw Time for TEST #3:		120.9	Raw Time for CONTROL #3:	153.4			
-13.8	O	1	ENUMERATION				
Raw Time for TEST #1:		95.3	Raw Time for CONTROL #1:	120.7			
Raw Time for TEST #2:		124.8	Raw Time for CONTROL #2:	118.1			
Raw Time for TEST #3:		104.8	Raw Time for CONTROL #3:	130.2			
-22.8	I_O	1	ENUMERATION				
Raw Time for TEST #1:		113.2	Raw Time for CONTROL #1:	84.5			
Raw Time for TEST #2:		96.7	Raw Time for CONTROL #2:	111.8			
Raw Time for TEST #3:		116.8	Raw Time for CONTROL #3:	110.3			
12.2	I	10	ENUMERATION				
Raw Time for TEST #1:		154.1	Raw Time for CONTROL #1:	112.4			
Raw Time for TEST #2:		139.5	Raw Time for CONTROL #2:	148.8			
Raw Time for TEST #3:		147.3	Raw Time for CONTROL #3:	160.1			
27.1	O	10	ENUMERATION				
Raw Time for TEST #1:		290.8	Raw Time for CONTROL #1:	226.6			
Raw Time for TEST #2:		268.8	Raw Time for CONTROL #2:	247.1			
Raw Time for TEST #3:		292.3	Raw Time for CONTROL #3:	246.6			
42.2	I_O	10	ENUMERATION				

tm_run.log

```
Number of Iterations = 10000


          TIME and DURATION math
          ======================
uSEC.   Operation
Raw test time     (seconds):     3.20     3.37     3.61     3.29     3.11
Raw control time (seconds):     0.93     1.13     1.01     0.96     0.91
  220.00   Time := Var_time + Var_duration
Raw test time     (seconds):     2.47     2.37     2.60     2.21     2.06
Raw control time (seconds):     1.30     1.12     0.98     1.01     0.99
  108.00   Time := Var_time - Var_duration
Raw test time     (seconds):     2.33     2.61     2.58     2.37     2.23
Raw control time (seconds):     1.03     1.34     1.24     0.94     0.99
  129.00   Time := Var_duration + Var_time
Raw test time     (seconds):     2.68     2.70     2.99     2.03     2.25
Raw control time (seconds):     1.00     1.27     1.23     1.12     0.89
  114.00   Time := Var_time + Const_duration
Raw test time     (seconds):     2.97     3.20     2.98     2.74     3.19
Raw control time (seconds):     1.21     0.92     0.92     1.07     1.14
  182.00   Time := Var_time - Const_duration
Raw test time     (seconds):     3.77     3.63     3.57     2.72     2.68
Raw control time (seconds):     1.27     0.90     1.06     0.91     0.94
  178.00   Time := Const_duration + Var_time
Raw test time     (seconds):     2.49     2.20     2.22     2.43     2.69
Raw control time (seconds):     1.95     1.75     1.54     1.51     1.89
   69.00   Duration := Var_time - Var_time
Raw test time     (seconds):     1.48     1.31     1.55     1.22     1.59
Raw control time (seconds):     1.56     1.22     1.25     2.02     1.76
    0.00   Duration := var_duration + var_duration
Raw test time     (seconds):     0.96     1.04     1.19     1.15     1.06
Raw control time (seconds):     1.10     0.94     0.91     1.41     1.28
    5.00   Duration := Var_duration - Var_duration
Raw test time     (seconds):     0.97     1.02     1.56     1.06     0.93
Raw control time (seconds):     0.97     0.94     1.10     1.10     0.96
   -0.99   Duration := Var_duration + Const_duration
Raw test time     (seconds):     1.54     1.54     1.53     1.30     1.31
Raw control time (seconds):     1.12     1.01     1.19     1.01     1.05
   29.00   Duration := Var_duration - Const_duration
Raw test time     (seconds):     1.22     1.30     1.64     1.23     1.19
Raw control time (seconds):     1.41     1.46     1.30     1.57     1.29
  -10.00   Duration := Const_duration + Var_duration
Raw test time     (seconds):     0.88     0.88     0.94     0.88     0.96
Raw control time (seconds):     1.11     1.02     1.13     1.02     1.09
```

```
  -14.00   Duration := Const_duration - Var_duration
Raw test time     (seconds):    0.94    1.06    1.03    1.12    1.14
Raw control time  (seconds):    1.11    1.34    1.11    1.29    0.91
    3.00   Duration := Const_duration + Const_duration
Raw test time     (seconds):    1.17    0.93    0.98    0.93    0.97
Raw control time  (seconds):    1.21    0.94    0.96    0.95    0.98
   -0.99   Duration := Const_duration - Const_duration
```

THIS IS A BLANK PAGE

Appendix III.
PIWG Analysis Program Output.

PIWG Analysis Report - produced on 12/7/1989.

Test Name	Time being Analyzed	Repeatable (Yes/No)	Best REPEATABLE Time	
A90	Clock Res	Yes	0.000061	Sec
A91	Time for 1 Dhry	Yes	0.897000	mSec
A93	Time per cycle	Yes	786.790000	mSec
A94	Perm	Yes	2.000000	Sec
A94	Towers	Yes	3.120000	Sec
A94	Queens	Yes	1.160000	Sec
A94	Intmm	Yes	1.070000	Sec
A94	Mm	Yes	1.080000	Sec
A94	Puzzle	Yes	9.010000	Sec
A94	Quick	Yes	1.030000	Sec
A94	Bubble	Yes	1.580000	Sec
A94	Tree	Yes	1.790000	Sec
A94	FFT	Yes	1.820000	Sec
A94	Ack	Yes	71.530000	Sec
B1	No suppress	Yes	63.060000	Sec
B1	Suppress	Yes	47.260000	Sec
B2	No suppress	Yes	27.380000	Sec
B2	Suppress	Yes	18.360000	Sec
B3	No suppress	Yes	44.360000	Sec
B3	Suppress	Yes	34.150000	Sec
B4	No suppress	Yes	32.850000	Sec
B4	Suppress	Yes	19.470000	Sec
C1	CPU time	Yes	7600.100000	uSec
C2	CPU time	Yes	7450.000000	uSec
C3	CPU time	Yes	7599.800000	uSec
D1	CPU time	Yes	15.600000	uSec
D2	CPU time	Yes	3699.800000	uSec
D3	CPU time	Yes	14.100000	uSec
D4	CPU time	Yes	5249.600000	uSec
E1	CPU time	Yes	562.500000	uSec
E2	CPU time	Yes	818.700000	uSec
E3	CPU time	Yes	681.300000	uSec
E4	CPU time	Yes	650.000000	uSec
E5	CPU time	Yes	349 .900000	uSec
F1	CPU time	Yes	21.900000	uSec
F2	CPU time	Yes	0.000000	uSec

G1	CPU time	Yes		1625.000000 uSec
G2	CPU time	Yes		5906.300000 uSec
G3	CPU time	Yes		3906.300000 uSec
G4	CPU time	Yes		5843.700000 uSec
G5	CPU time	Yes		375.000000 uSec
G6	CPU time	Yes		1078.100000 uSec
G7	CPU time	Yes		54998.800000 uSec
H1	CPU time	Yes		0.400000 uSec
H2	CPU time	Yes		900.000000 uSec
H3	CPU time	Yes		812.500000 uSec
H4	CPU time	Yes		278.100000 uSec
H5	CPU time	Yes		0.000000 uSec
H6	CPU time	Yes		29.700000 uSec
H7	CPU time	Yes		63.300000 uSec
H8	CPU time	Yes		24.200000 uSec
H9	CPU time	Yes		70.300000 uSec
L1	CPU time	No	*	
L2	CPU time	No	*	
L3	CPU time	Yes		0.000000 uSec
L4	CPU time	Yes		0.000000 uSec
L5	CPU time	Yes		13.400000 uSec
P1	CPU time	Yes		0.400000 uSec
P2	CPU time	Yes		28.900000 uSec
P3	CPU time	Yes		27.000000 uSec
P4	CPU time	Yes		0.000000 uSec
P5	CPU time	Yes		27.700000 uSec
P6	CPU time	Yes		29.700000 uSec
P7	CPU time	Yes		31.300000 uSec
P10	CPU time	Yes		50.800000 uSec
P11	CPU time	Yes		85.900000 uSec
P12	CPU time	Yes		43.700000 uSec
P13	CPU time	Yes		60.200000 uSec
T1	CPU time	Yes		1181.200000 uSec
T2	CPU time	Yes		1349.900000 uSec
T3	CPU time	Yes		1250.000000 uSec
T4	CPU time	Yes		1362.500000 uSec
T5	CPU time	Yes		1270.000000 uSec
T6	CPU time	Yes		2059.900000 uSec
T8	CPU time	Yes		2825.000000 uSec

Appendix IV.
Contents of PIWG and UMICH suites.

PIWG Suite.

The PIWG suite consists of approximately 230 files which are arranged as:

- 1 README file. This file provides details of how the suite is arranged and some basic information on how to perform the tests.
- Approximately 205 Ada source files. These are the actual tests. The README file indicates what each file does.
- Approximately 20 command files. Command files are provided for different platforms (e.g., .BAT files are provided for PCs and .COM files for VAX/VMS environments). These files may need to be modified if the tests are to be run with different compilers.
- 1 sample output file. This file shows the output produced by the tests.
- 2 distribution information files. One of these file contains a "results form" which should be completed and forwarded to PIWG by the users of the suite so that the central PIWG database can be updated with the test results. The other file contains a tape distribution tree which contains the names and addresses of distributors of the tape.

UMICH Suite.

The UMICH suite consists of approximately 200 files which are made up of:

- 1 README file. This file provides an explanation of the tests. A reference to a report that discusses timing issues, rationale, and results for the Verdix Ada compiler is also given.
- Approximately 160 Ada source files. These are the actual tests. The README file explains the naming conventions used for these files.
- Approximately 30 command files. These command files contain the necessary commands to compile and run the tests. Once again, modification of these files will be needed if the tests are to be run on different platforms.
- 1 main unit file list file. This file lists the names of the main procedures of the tests along with the name of the files in which their source code is contained.

THIS IS A BLANK PAGE

Glossary.

ACEC — Ada Compiler Evaluation Capability

ACM — Association for Computing Machinery

AdaIC — Ada Information Clearing house

AES — Ada Evaluation System

AJPO — Ada Joint Program Office

APSE — Ada Programming Support Environment

ASR — Ada Software Repository

BGT — Benchmark Generator Tool

BRT — Best Repeatable Time

DDN — Defence Data Network

DEC — Digital Equipment Corporation

E&V — Evaluation & Validation

IDA — Institute for Defence Analyses

PIWG — Performance Issues Working Group

SD — Systems Designers

SEI — Software Engineering Institute

SIGAda — Special Interest Group on Ada

UMICH — University of MICHigan

VAX — 32 bit computer manufactured by DEC

THIS IS A BLANK PAGE

DISTRIBUTION

	Copy No.
DEPARTMENT OF DEFENCE	
Defence Science and Technology Organisation	
Chief Defence Scientist	1
First Assistant Secretary Science Policy (FASSP)	
Counsellor Defence Science Washington	Cnt Sht Only
Counsellor Defence Science London	Cnt Sht Only
Electronics Research Laboratory	
Director, Electronics Research Laboratory	2
Chief, Information Technology Division	3
Head, Software Engineering	4
Publicity and Component Support Officer, Information Technology Division	5
Joint Intelligence Organisation (DSTI)	6
Libraries and Information Services	
Librarian, Technical Reports Centre, Defence Central Library, Campbell Park	7
Document Exchange Centre Defence Information Services and Science Liaison Branch for:	
Microfiche copying	8
United Kingdom, Defence Technical Information Center	9 - 10
United States, Defense Technical Infomration Center	11 - 22
Canada, Director, Scientific Information Services	23

DOCUMENT CONTROL DATA SHEET

Security classification of this page : UNCLASSIFIED

1 DOCUMENT NUMBERS

AR
Number : AR-006-405

Series
Number : ERL-0513-TR

Other
Numbers :

2 SECURITY CLASSIFICATION

a. Complete Document : Unclassified

b. Title in Isolation : Unclassified

c. Summary in Isolation : Unclassified

3 DOWNGRADING / DELIMITING INSTRUCTIONS

Limitation to be reviewed in April 1993

4 TITLE

ADA COMPILER BENCHMARKING: INITIAL INVESTIGATIONS.

5 PERSONAL AUTHOR (S)

R. Vernik
I. Turner
S.F. Landherr

6 DOCUMENT DATE

April 1990

7 7.1 TOTAL NUMBER OF PAGES 43

7.2 NUMBER OF REFERENCES 16

8 8.1 CORPORATE AUTHOR (S)

Electronics Research Laboratory

8.2 DOCUMENT SERIES and NUMBER

Technical Report 0513

9 REFERENCE NUMBERS

a. Task : DST 89/047

b. Sponsoring Agency :

10 COST CODE

223AA254

11 IMPRINT (Publishing organisation)

Defence Science and Technology Organisation Salisbury

12 COMPUTER PROGRAM (S) (Title (s) and language (s))

13 RELEASE LIMITATIONS (of the document)

Approved for Public Release.

Security classification of this page : UNCLASSIFIED

Security classification of this page : UNCLASSIFIED

14 ANNOUNCEMENT LIMITATIONS (of the information on these pages)

No limitation

15 DESCRIPTORS

a. EJC Thesaurus Terms

Ada programming language, Compilers, Benchtests. (CP)

b. Non - Thesaurus Terms

16 COSATI CODES

1205

17 SUMMARY OR ABSTRACT

(if this is security classified, the announcement of this report will be similarly classified)

This paper documents some initial investigations into the benchmarking of Ada compilers. A summary of available benchmarking suites is given, although only two of these suites were used in the initial benchmarking experiments: the ACM SIGAda Performance Issues Working Group ~~(PIWG)~~ benchmarks and the University of Michigan ~~(UMICH)~~ benchmarks. Experiences and lessons learned in applying these suites to the Alsys Ada compiler hosted on a Toshiba personal computer and to the DEC VAX Ada compiler hosted on a VAX 8300 are provided. Based on these initial benchmarking experiences, several areas of possible further research/development are identified. In particular, the need for more advanced analysis tools is discussed. Keywords: Ada

Security classification of this page : UNCLASSIFIED