SOFTWARE ENGINEERING LABORATORY (SEL)

ADA PERFORMANCE STUDY REPORT

JULY 1991

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-91-003

SOFTWARE ENGINEERING LABORATORY (SEL)

ADA PERFORMANCE STUDY REPORT

JULY 1991

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

- NASA/GSFC, Systems Development Branch
- The University of Maryland, Computer Sciences Department
- Computer Sciences Corporation, Systems Development Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environments; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The major contributors to this document are

Eric W. Booth (CSC)
Michael E. Stark (GSFC)

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ABSTRACT

This document describes the goals of the Ada Performance Study, explains the methods used, presents and analyzes the results, and provides guidelines for future Ada development efforts. Section 1 details the goals and scope of the study and presents the background of Ada development in the Flight Dynamics Division (FDD). Section 2 presents the organization and overall purpose of each test. Section 3 details the purpose, methods, and results of each test and provides analyses of these results. Section 4 provides guidelines for future development efforts based on the analysis of results from this study. Appendix A explains the approach used on the performance tests.

Table of Contents

Table of Contents (Cont'd)

List of Illustrations

List of Tables

EXECUTIVE SUMMARY

INTRODUCTION

The need to predict, measure, and control the run-time performance of systems in the Flight Dynamics Division (FDD) is a growing concern as software systems become more sophisticated. The transition to Ada introduces performance issues that were previously nonexistent. Additionally, this transition to Ada was accompanied by the transition to object-oriented development (OOD), which has performance implications independent of the programming language. To better understand the implications of new design and implementation approaches, the Software Engineering Laboratory (SEL) conducted the Ada Performance Study.

OBJECTIVES AND SCOPE

The Ada Performance Study had the following three objectives:

- Determine which design and implementation alternatives lead to accelerated run times
- Determine what, if any, trade-offs are made by choosing these alternatives
- Develop guidelines to aid future Ada development efforts in the FDD

The study focused on the run-time performance of existing Ada and OOD approaches used in the FDD. The study did not address compiler, linker, or other development tool performance.

RESULTS

The following statements summarize the results of the Ada performance study:

- Incorrect design decisions were the largest contributor to poor run-time performance. The design should continually be reevaluated against evolving user requirements and specifications.
- Current Ada compilation systems still have inconvenient bugs that may contribute to poor performance. Organizations using Ada should use available performance analysis tools to assess their compilation systems.
- Ada simulators in the FDD can be designed and implemented to achieve run times comparable to existing FORTRAN simulators. Inefficient systems indicate errors in the system design and/or the compiler being used.

As these results indicate, few, if any, trade-offs were necessary to achieve accelerated run-times comparable to existing FORTRAN simulators. This report contains a

detailed analysis of each alternative studied and summarizes the results of this analysis with specific performance recommendations for future OOD/Ada development efforts in the FDD.

SECTION 1 – INTRODUCTION

The Ada language reference manual (LRM) (Reference 1) states:

> Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency.

Initial implementations of Ada compilers and development environments tended to favor the first two concerns over the concern for efficiency. Similarly, initial (non-real-time, non-embedded) applications development using Ada as the programming language tended to favor maintainability, readability, and reusability.

The need to predict, measure, and control the run-time performance of systems in the Flight Dynamics Division (FDD) is a growing concern as software systems become more sophisticated. The transition to Ada introduces performance issues that were previously nonexistent. Additionally, this transition to Ada was accompanied by the transition to object-oriented development (OOD), which has performance implications independent of the programming language. To better understand the implications of new design and implementation approaches, the Software Engineering Laboratory (SEL) conducted the Ada Performance Study.

1.1 OBJECTIVES AND SCOPE

The Ada Performance Study had the following three objectives:

- Determine which design and implementation alternatives lead to accelerated run times
- Determine what, if any, trade-offs are made by choosing these alternatives
- Develop guidelines to aid future Ada development efforts in the FDD

The study focused on the run-time performance of existing Ada and OOD approaches used in the FDD. The study did not address compiler, linker, or other development tool performance.

The entire study was performed on a Digital Equipment Corporation (DEC) VAX 8820 machine running VMS version 5.2 and DEC Ada version 1.5-44.

The performance tests were developed using a commercial performance measurement tool on actual flight dynamics software systems in combination with a tailored Association for Computing Machinery (ACM) special interest group on Ada (SIGAda) performance issues working group (PIWG) structure of measurements. The commercial performance measurement tool used during the study was DEC's performance coverage analyzer (PCA). Appendix A explains, in detail, how the PIWG structure of measurements and PCA were used during the Ada Performance Study.

1.2 BACKGROUND OF ADA IN THE FDD

The SEL started experimenting with Ada and object-oriented approaches in 1985 with the Gamma Ray Observatory (GRO) attitude dynamics simulator in Ada (GRODY) experiment. This experiment included an electronic mail system (EMS) training project and parallel development of the operational GRO dynamics simulator in FORTRAN (GROSS). GROSS was developed using a traditional functional decomposition design and FORTRAN implementation. Figure 1-1 shows the timeline of the FDD Ada development efforts. The GRODY team was encouraged to experiment with new methodologies and techniques and used an OOD-Ada approach. As a result, several lessons were learned about OOD methods (Reference 2), implementation approaches, and testing techniques (Reference 3).

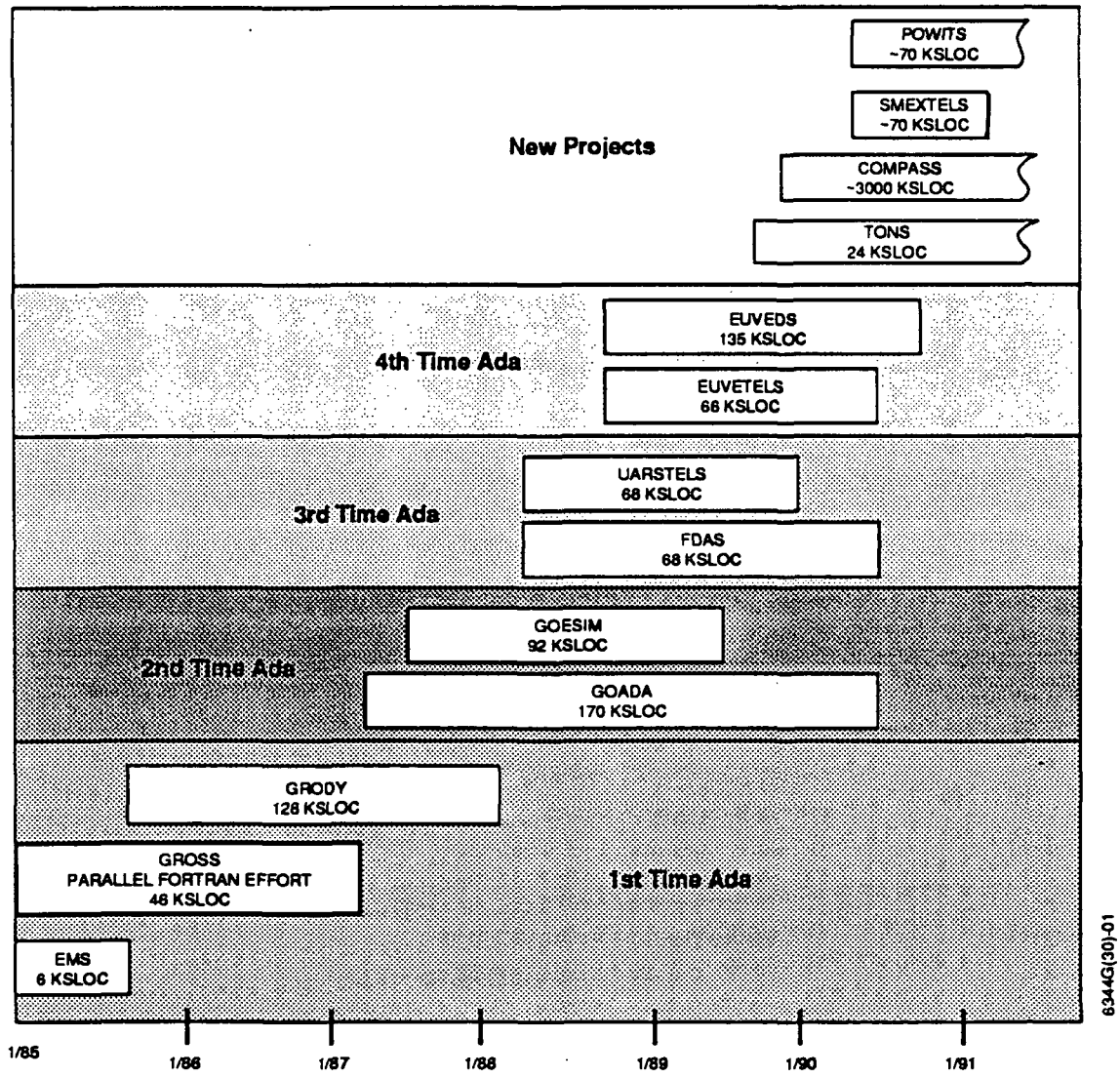

Figure 1-1. Timeline of Ada Development in the FDD

Following the GRODY experiment, the FDD began support for the Geostationary Operational Environmental Satellite (GOES) with the GOES attitude dynamics simulator in Ada (GOADA) and the GOES telemetry simulator (GOESIM). These simulators incorporated the lessons learned from the GRODY experiment and were developed using OOD and Ada. Driven by the promise of increased reusability with Ada, GOADA and GOESIM simulator teams performed an in-depth reuse analysis of GRODY. As a result, many design decisions and software components developed during the GRODY experiment were incorporated and reused.

This trend of reusing design and code has continued throughout the development of the OOD-Ada simulators in the FDD with extremely promising results (see Figure 1-2). Software development costs have been reduced by as much as 66 percent, and software reliability has improved by an order of magnitude. However, a side effect of this emphasis on software reuse has been the proliferation of inefficient design approaches and implementation techniques.

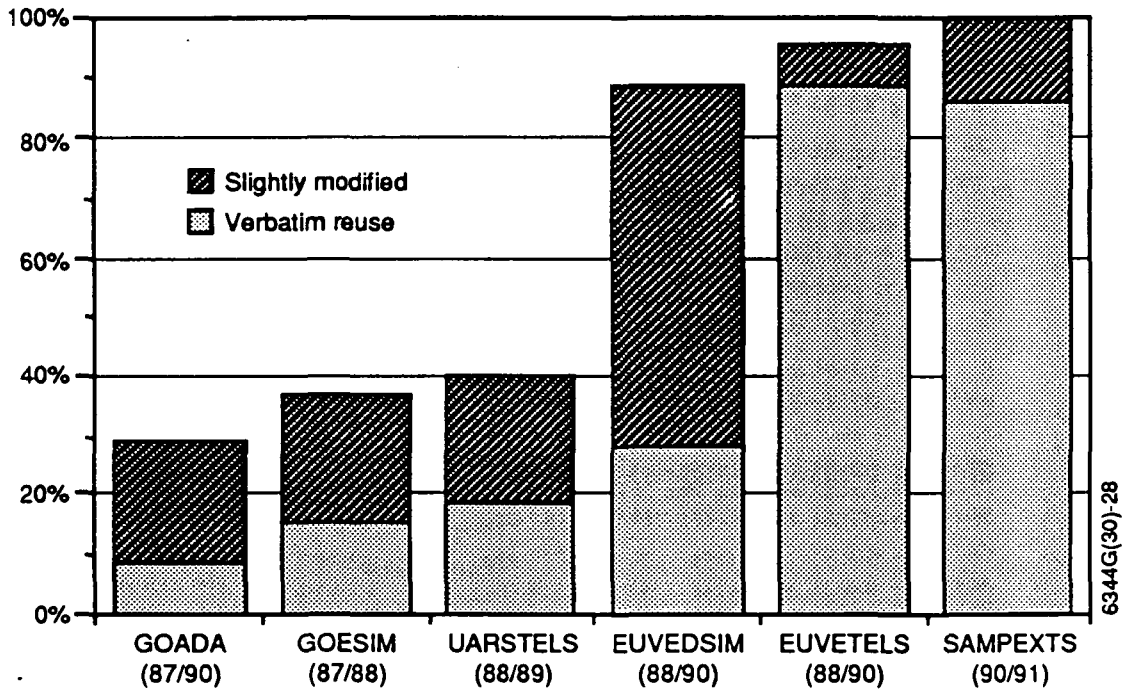

Figure 1-2. Software Reuse Trend in the FDD Using OOD and Ada

As a result, each of the OOD-Ada simulators incorporate fundamental design and implementation inefficiencies. The Ada Performance Study investigated design and implementation approaches that were known to contribute or suspected of contributing to the largest of the observed inefficiencies. Each of these approaches was measured and compared with other, potentially more optimal, approaches. The approaches offering improvements, as well as those lacking improvement, were

recorded. This report summarizes these results to provide guidance for future OOD-Ada designers and to predict more accurately achievable run-time performance.

1.3 DOCUMENT OVERVIEW

This document is organized into the following sections:

- *Section 2—Test Overview* presents the organization and purpose of the different test groups.
- *Section 3—Results* details the purpose of, methods used during, results from, and analysis of each performance test.
- *Section 4—Conclusions* provides performance guidelines for future OOD-Ada development projects in the FDD.
- *Appendix A—Approach to Measurement* presents the two approaches used during the performance study and describes the details of each.

SECTION 2 – TEST OVERVIEW

Ten test groups were developed; each represented a design or implementation issue that is relevant to current FDD applications. The test groups are presented in two categories: design-oriented tests and implementation-oriented tests. The test groups were chosen as a result of an in-depth analysis of several PCA runs with two simulators: GOADA and the Extreme Ultraviolet Explorer (EUVE) telemetry simulator (EUVETELS). If a certain design alternative or language feature appeared to consume a relatively large portion of central processing unit (CPU) time or memory, it was analyzed, measured, and quantified in this study. The design alternatives or language features consuming a relatively small portion of CPU time or memory were not studied further. Therefore, the test groups presented here are intended as a representative rather than exhaustive sampling of current design and implementation approaches.

2.1 DESIGN-ORIENTED TESTS

The following briefly describes the purpose of each design test group run by this study.

- *Group 1: Scheduling*—This test group contained three tests that addressed the run-time cost of various scheduling alternatives. In one test case, user flexibility and result accuracy were sacrificed to achieve more optimal performance. The implications of the different approaches are analyzed and contrasted. The results of this test group provided the applications designers with information necessary to make trade-off decisions among flexibility, accuracy, and performance.

- *Group 2: Unconstrained Structures*—Leaving data structures unconstrained provides greater user flexibility and enhances future reusability. However, the additional run-time code that may be generated can impose a significant run-time and memory overhead. This group measured the expense of unconstrained records and arrays and proposed viable alternatives.

- *Group 3: Initialization and Elaboration*—This test group addressed initialization of static and dynamic data using various combinations of elaboration- and execution-time alternatives. This test group was relevant for applications requiring minimal initialization time.

- *Group 4: Generic Units*—The benefits of using generic units are reduced source code size, encapsulation, information hiding, decoupling, and increased reuse (Reference 4). However, many Ada compilers implement this language feature poorly. This test group addressed the options available with the compiler implementation and examined how well these options were implemented.

- *Group 5: Conditional Compilation*—The ability to include additional "debug code" in the delivered system adds to the system size and imposes a run-time

penalty even if it is never used. The test group analyzed the current approach and proposed flexible alternatives for future systems. The results of this test group can have applications beyond debug code elimination.

- *Group 6: Object-Oriented Programming*—Two of the fundamental principles of object-oriented programming (OOP) are polymorphism and inheritance. Ada does not directly support these principles. However, the designer may simulate the effect of inheritance and polymorphism through the use of variant records and enumeration types (Reference 5). These OOP principles, whether direct or indirect, incur a certain amount of run-time overhead and problems (Reference 5).

2.2 IMPLEMENTATION-ORIENTED TESTS

The following briefly describes the purpose of each implementation test group run by this study.

- *Group 7: Matrix Storage*—The most basic and perhaps the most common mathematical expressions in flight dynamics applications involve matrix manipulations. This group addressed row-major versus column-major algorithms to quantify the performance implications.

- *Group 8: Logical Operators*—The Ada LRM clearly defines behavior of logical expression evaluation. The Ada Style Guide (Reference 6) recommends not using the short circuit forms of logical operators for performance reasons. The implications of this recommendation in the flight dynamics environment were measured and analyzed.

- *Group 9:* ***pragma*** *INLINE*—Flight dynamics simulators contain a large number of procedure and function calls to simple call-throughs and selectors. The overhead of making these calls can slow the performance of any simulator. This test measured the use of **pragma** INLINE as an alternative to calling a routine.

- *Group 10: String-to-Enumeration Conversion*—Current flight dynamics simulators contain a central logical data base. The physical data are distributed throughout the simulator in the appropriate packages. The logical data base provides keys (strings) that map into the physical data. The logical data base converts these strings to the appropriate enumeration type to retrieve the corresponding data. This test assessed the performance implications of this approach.

SECTION 3—RESULTS

Each performance test in this report is described in this section in the following format:

- *Purpose*—Each test was designed with a specific design or implementation alternative in mind. Rationale for the choice of alternatives tested results from analysis of existing Ada systems developed in the FDD.

- *Method*—Some tests were performed as changes to an existing system, whereas other tests were performed by creating new, special-purpose software. Methods used for each test are described in detail. The basis for each method consisted of one of two approaches: DEC's PCA measurement tool or the PIWG structure of measurements. Appendix A contains a detailed discussion of the two approaches.

- *Results*—The result of executing a test is some combination of CPU time and object code size. Most tests were designed to measure the CPU run time in microseconds (μs). In some cases the object code size in bytes is relevant. The data resulting from each test run are provided.

- *Analysis*—In many cases, detailed analysis of the test results is necessary to understand the implications for future projects. The analysis performed is summarized, and the implications are highlighted.

3.1 DESIGN-ORIENTED TESTS

3.1.1 Scheduling

3.1.1.1 PURPOSE

Flight dynamics simulators all have a method for performing simulation control. Most FORTRAN-based simulators used a time-driven, hard-coded approach. The algorithm for this approach is shown in Figure 3-1.

The loop control variable for this time-driven algorithm is Current_Time. All events (e.g., Model CSS1) occur at the same simulation time. Finally, all events are hard-coded; that is, the models that are called and the order in which they are called are determined at compilation time. The CPU time associated with this design is the cost of incrementing and testing the simulation time.

The approach used for most of the Ada/OOD-based simulators is more of an event-driven design. The algorithm for this approach is shown in Figure 3-2.

The loop control variable for this event-driven algorithm is Current_Event. The CPU time associated with this design is the cost of setting the simulation clock, dispatching

```
Current_Time :=Start_Time;
while Current_Time ≤ Stop_Time loop
   Advance_Ephemeris (To => Current_Time);
   Advance_Attitude (To => Current_Time);
   Model_CSS1;
   Model_FSS;
   ... --model all remaining sensors
   Model_OBC;
   ... --model all actuators
   Current_Time := Current_Time + Time_Step;
end loop;
```

Figure 3-1. Time-Driven, Hard-Coded Algorithm

```
Current_Event := Next_Event (On_the => Queue);
while Current_Event ≠ Stop_Simulation loop
   Set_the_Clock (To => Current_Event Time);
   Dispatch (Current_Event);
   Reschedule (Current_Event, On_the => Queue);
   Current_Event := Next_Event (On_the => Queue);
end loop;
```

6344G(30)-02

Figure 3-2. Event-Driven Algorithm

the event, rescheduling the event, popping the next event off the queue, and testing the event. Although the CPU time necessary for this event-driven algorithm will be greater than that necessary for the time-driven algorithm, the choice of the algorithm really depends on two items: the characteristics of the data and the desired flexibility of the final system.

The time-driven approach favors uniformly distributed data. That is, each event is modeled at the same time for the same frequency throughout the simulation. The event-driven approach favors random, even erratic, data. For example, some sensor data may be modeled frequently [e.g., every 32 milliseconds (ms)], whereas other sensor data might be modeled infrequently (e.g., every 64 seconds).

The hard-coded approach of the time-driven algorithm requires little run-time overhead. However, this is also a very inflexible design. Adding, deleting, or reordering the events requires changes to the source code.

The event-driven algorithm imposes a higher run-time overhead. However, infrequently occurring events in the real world are modeled infrequently in the simulator, thereby reducing CPU time. Event-driven approaches also provide a level of flexibility to the design. Adding a previously nonexistent event would still require

modifications to the source code. However, re-adding, deleting, and reordering events may be accomplished without source code changes.

This test group contained three tests that addressed the run-time cost of various scheduling alternatives. Flexibility and accuracy were sacrificed to achieve more optimal performance. The implications of the different approaches were analyzed and compared. The results of this test group provided the applications designers with information necessary to make trade-off decisions among flexibility, accuracy, and performance.

3.1.1.2 METHOD

The test group was set up and run in the context of the baseline version of the GOADA dynamics simulator. The output from the batch log file and the PCA was used to assess the impact of alternate designs. The baseline version of the scheduler in GOADA was measured first. Two design alternatives of the scheduler were tested: the first modified the internal data storage and the second modified the scheduling algorithm.

The first design alternative involved using a different data structure for scheduling events in a discrete event simulation. The delivered version of the scheduler for GOADA uses an array data structure to maintain the queue. The alternate version uses a linked-list data structure constructed with access types. This test was designed to compare the efficiency of the insertion operation of an array data structure with the same operation on a linked-list data structure.

The second design alternative tested a looping algorithm to dispatch events in a simulator. The delivered version of the scheduler for GOADA is event-driven. The simulation time is set by the value of the next event on the priority queue. However, after a modification late in the development phase, all events fired at the set interval of 512 ms. The alternate version tested uses a time-driven algorithm that iterates over a static queue of events.

3.1.1.3 RESULTS

As shown in Table 3-1, the first design alternative had no measurable effect on performance. It was determined that the CPU time saved by not sliding the elements of the array during an insertion is offset by stepping though a linked list during a search. This implied that sliding a slice of an array is a relatively efficient operation. Analysis of the assembler code generated by the compiler revealed that DEC Ada performs a "block move" operation when assigning a slice of an array.

3.1.1.4 ANALYSIS

The second design alternative shows a 25-second speed-up for a 20-minute simulation. For comparison, the original Scheduler package consumed 10.7 percent of the CPU time needed (6 minutes, 45 seconds) to run GOADA for a 20-minute simulation. The

Table 3-1. Results From Scheduling Design Alternatives

Variables	Baseline Version of GOADA	First Alternative: Linked-List	Second Approach: Iterative Algorithm
Total CPU Time	6:45	6:45	6:20
Scheduler Percentage	10.7%	11%	2.2%
Scheduler CPU Time	44 seconds	45 seconds	8.5 seconds

6344G(30)-03

improved Scheduler package requires only 2.2 percent of the total CPU time (6 minutes, 20 seconds). Table 3-1 summarizes the results of this test.

The results of this test underscore two important rules when dealing with programming languages that allow the user to define abstract data types:

- During the preliminary design phase, choose the data structure that most closely matches the data for the problem being solved.
- During the detailed design phase, match the algorithm to both the data structure and the data.

During the design phase, the GOADA developers correctly matched the data structure (a priority queue), the data (discrete, time-ordered events), and the algorithm. Late in the GOADA development life cycle, however, the data that were stored in the priority queue were modified to the extent that the priority queue data structure and the algorithm no longer matched. The following lesson was learned from this experience:

- Modifications made to an Ada system during the testing and maintenance phases must address both algorithmic and data structure changes to ensure that they both still match the problem being solved.

3.1.2 Unconstrained Structures

This section addresses the use of unconstrained types, both records and arrays, in a flight dynamics application. Two separate test approaches were used to assess the run-time cost of unconstrained data types. The first test addresses the use of an unconstrained variant record used in the GOADA simulator. The second test addresses the use of unconstrained versus constrained matrixes.

3.1.2.1 VARIANT RECORDS

3.1.2.1.1 Purpose

Current flight dynamics simulators make extensive use of an unconstrained variant record structure. This approach allowed a general communication interface between the user interface subsystem and the simulation subsystem. It was expected that this

approach led to increased access time as compared with a constrained (nonvariant) record structure. This test measures the overhead associated with using this type of data structure.

3.1.2.1.2 Method

The hardware and environment models in GOADA each declare their state data as an array of these variant records. It was expected that the use of the variant record data structure throughout the simulator was causing a performance penalty that is distributed across many packages (and thus, not highlighted by PCA).

The Earth sensor assembly (ESA) package was chosen as the basis for this test because it is the most CPU-expensive hardware model in GOADA that uses the variant record structure. The internal data structure was modified to be a static, rather than dynamic, record. The subprograms that previously accessed an element of the array were modified to access the corresponding field of the record. External communication with the user interface mapped the array elements to the appropriate record field using a case statement.

3.1.2.1.3 Results

As shown in Table 3-2, the percentage of CPU time consumed by the ESA dropped from 3.7 to 1.1 percent. This means that the change to the data structure used to store the ESA data decreased the CPU time to less than one-third of its previous amount.

Table 3-2. Results From ESA Record Structure Modification

Processing	Baseline Version of GOADA	Static Record Structure
Total CPU Time	6:45	6:33
ESA Percentage	3.7%	1.1%
ESA CPU Time	15 seconds	4.3 seconds

6344G(30)-04

3.1.2.1.4 Analysis

Alone, the ESA improvement is minor (the total speed-up for the 6-minute, 45-second benchmark was about 11 seconds), but analysis of the PCA output and the source reveals that 45 percent of the CPU time is spent in packages using the same variant data structure as the ESA. Assuming that altering the data structure in all of these remaining packages has a similar effect, the 45 percent of the CPU time could be reduced to 13 percent. This would mean an overall decrease in the total CPU time by 32 percent.

The results and analysis of this test have many implications. Data structures used for external communication with an input/output (I/O) device or user interface may be

sufficiently general to encourage reusability. However, the internal data structures should be optimal for the problem domain. This may be stated as follows:

- The state data for a package should use a data structure that matches the data and algorithms being used (see test group 1, Scheduling). Sufficiently general constructors and selectors may be designed that map data of a general type to the optimal state data in the package body.

This approach sacrifices some generality to achieve efficiency. On the other hand, the explicitness required in the design and implementation of the constructors and selectors encourages the developer to exploit the strong typing feature of the language.

3.1.2.2 MATRIXES

3.1.2.2.1 Purpose

Flight dynamics simulators contain a large number of two-dimensional arrays. The varying ways of accessing and constraining these arrays can affect the addressing performance. Three approaches may be applied to two-dimensional array (matrix) operations that may affect performance:

- Use of a literal range for constrained and unconstrained matrixes
- Use of the 'range attribute for constrained and unconstrained matrixes
- Use of the **pragma** Suppress_All for constrained and unconstrained matrixes

Various combinations of each of these three approaches were examined. [NOTE: DEC Ada does not support **pragma** Suppress. Instead, DEC provides an implementation-specific **pragma** Suppress_All.]

3.1.2.2.2 Method

The first technique assessed the cost of using a literal range to reference a matrix (both constrained and unconstrained). The second performance technique assessed the cost of using the range attribute to reference a matrix (both constrained and unconstrained). The last technique examined the DEC VAX-Ada-specific directive **pragma** Suppress_All. Suppress_All suppresses all run-time checking, including arrays. The code segments shown in Figure 3-3 highlight the different accessing methods tested.

This test comprises the following experiments:

- Referencing a constrained matrix using a literal range
- Referencing an unconstrained matrix using a literal range
- Referencing a constrained matrix using the range attribute

Literal Range Specified	Use of Range Attribute

for i in 1..6 loop
  for j in 1..6 loop
  Matrix(i,j) := i * j;
  end loop;
end loop;
	for i in Matrix'range (1) loop
	  for j in Matrix'range(2) loop
	  Matrix(i,j) := i * j;
	  end loop;
	end loop;

6344G(30)-05

Figure 3-3. Simple Code Segment From the Matrix Access Test

- Referencing an unconstrained matrix using the range attribute
- Finally, repeating all four experiments using **pragma** Suppress_All

3.1.2.2.3 Results

This performance test must be examined in a number of ways. The bar chart in Figure 3-4 contains the performance costs and gains of the various access techniques.

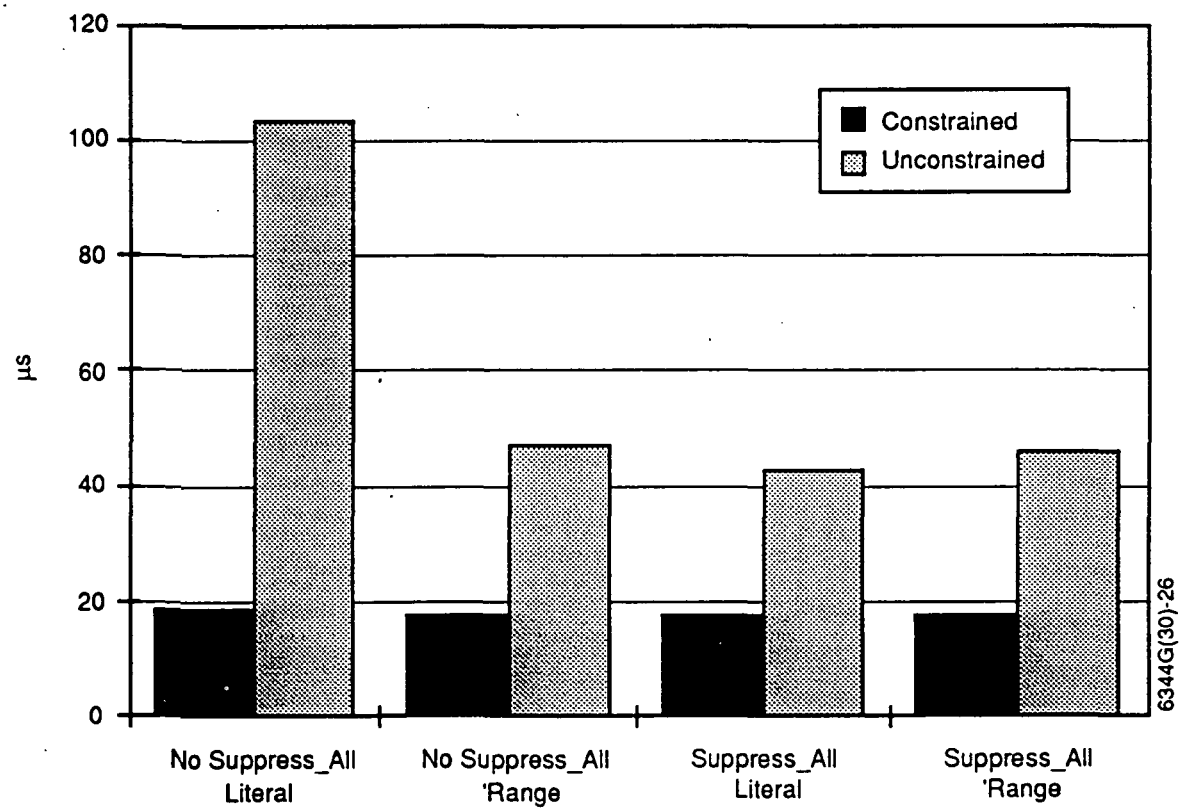

Figure 3-4. Comparison of Times (µs) for Matrix Access Methods Test

Figure 3-4 shows the dramatic difference between constrained and unconstrained matrixes. The CPU time when using an unconstrained array is double that when using a constrained matrix. Even with the use of **pragma** Suppress_All, the difference is large. The use of **pragma** Suppress_All had little effect on the range attribute address mechanism. The real gain in using Suppress_All came with the use of a literal range when accessing the matrix. The use of a literal range with **pragma** Suppress_All was close in performance to the use of the range attribute. The raw data for this test are shown in Table 3-3.

Table 3-3. Times for Matrix Access Methods Test

Reference Method	No Suppress (μs)	Suppress (μs)
Constrained matrix using a literal range	18.6	17.6
Unconstrained matrix using a literal range	103.4	43.3
Constrained matrix using the range attribute	17.8	17.9
Constrained matrix using the range attribute	47.5	46.4

6344G(30)-06

3.1.2.2.4 Analysis

Using **pragma** Suppress_All improves performance where there is range and access checking on arrays. This checking occurs when a literal range is specified in the processing of an unconstrained matrix. The sample code in Figure 3-3 shows that there is no way for the compiler to know whether the indexes that are being used are inside the bounds of the declared matrix. Therefore, range checking is being enforced, and this negatively impacts performance.

In Figure 3-3, use of the range attribute provides the compiler with enough information to eliminate run-time constraint checking code. The only cost is the initial evaluation of the 'range attribute. However, analysis of the assembler code generated from the source code shown in Figure 3-3 reveals that while the value of **'range(1)** is calculated only once, the (loop invariant) value of **'range(2)** is calculated within the outer loop. This represents a bug in the DEC Ada version 1.5-44 compiler. Similar compiler bugs have been previously discovered (Reference 7). DEC has corrected this problem in the 2.0 version of the Ada compiler.

The most obvious performance gain to be made in accessing a matrix is to constrain it. Unconstrained matrixes have an additional cost attributed to them due to their dynamic nature. The additional cost comes from the run-time checking necessary. The

generated range checking code also consumes additional memory. Use of the range attribute also eliminates run-time range checking, but the current version of the compiler generates inefficient object code. Finally, operations on a matrix of a constrained type may be checked at compilation time, because the bounds are static rather than dynamic. The implications of this test follow:

- Whenever the size of the structure (record or array) is truly static for a particular domain, define the type as a constrained type.
- Use attributes wherever possible when unconstrained structures are necessary.

3.1.3 Initialization and Elaboration

3.1.3.1 PURPOSE

This test group compared various methods of initializing objects of different types. The PCA revealed an unexpectedly large portion of CPU time spent performing elaboration. For example, objects were often initialized explicitly at subprogram elaboration even though unconditional initialization of objects occurred in the subprogram body. This test group consisted of four experiments, each of which compared two different initialization methods.

3.1.3.2 METHOD

This test was divided into four experiments containing two different initialization tests each, as follows:

- Elaboration versus assignments-in-the-body
- Array/record aggregates versus separate component assignments
- Package body elaboration code versus explicit call
- Types with default values

Each test was developed using the PIWG methodology. The method for each of these four experiments is outlined below.

The elaboration versus assignments-in-the-body experiment compared initializing objects of arbitrary types at subprogram elaboration time with initializing the same objects in a subprogram body via assignment statements. The subprogram elaboration test yielded the CPU time taken for initializing an integer object, a float object, a Boolean object, and a three vector of integers at subprogram elaboration time. The assignment test gave the CPU time to initialize the same objects using assignment statements in the procedure body.

The array/record aggregates versus separate component assignments experiment compared initializing objects of array and record types using record and array

aggregate assignments in the subprogram body with using separate component assignments in the subprogram body. Both tests used a record consisting of integer, float, and Boolean components and a three vector of integers.

The package body elaboration code versus explicit call experiment contrasted use of package body elaboration for initializing state data with use of an explicit call to an initialization procedure for the same purpose. To measure the time for package body elaboration, the elaboration test was designed to have a dynamic package created each time the test procedure was called; that is, a package that has elaboration code in the package body is declared inside the test procedure. In both tests, integer, float, and Boolean components and a three vector of integers were initialized.

The types with default values experiment determined the CPU time expense for having a record type with default values followed by initialization of an object of that type at subprogram elaboration time and via assignment statements in the subprogram body. The record contained two components of type float.

3.1.3.3 RESULTS

The results for each of the experiments are presented in Table 3-4. Analysis of these results is presented in the next subsection.

Table 3-4. CPU Time (μs) Results of the Various Initialization Tests

Elaboration	Assignment
1.8	1.7
Aggregate Assignment	Separate Component
3.0	3.3
Package Elaboration	Explicit Call
2.7	7.7
Type with Defaults Elaboration	Types with Defaults Assignment
0.0	1.0

6344G(30)-07

3.1.3.4 ANALYSIS

The CPU times associated with the first experiment (elaboration versus assignment in the body) indicate that the two approaches are effectively the same. Table 3-4 shows that there was no time penalty for initializing objects at subprogram elaboration over using assignments in the subprogram body.

The results of the second experiment (array/record aggregates versus separate component assignments) demonstrated approximately a 10-percent reduction in CPU time for using aggregate assignments.

As expected, the results of the third experiment (package body elaboration code versus explicit call) show that the cost of the explicit call was more expensive. Initializing state data via an explicit call (for this case) required approximately 2.85 times more CPU time than initializing state data at elaboration time.

The results of the fourth experiment (types with default values) showed that the CPU time for initializing the components of the record in the subprogram body took approximately 1.0 ms, which is simply the time required to do an assignment statement of the record aggregate. Initializing the object at subprogram elaboration did not incur the cost of this additional assignment. The assembly code for the tests showed that when an object of the record type with default values is declared, these default values get assigned to the object at this point. However, in the case of the elaboration test, the object does not get assigned the default value associated with the type, but instead the value is assigned to it in the object declaration.

The implications of this test are as follows:

- When dynamic data are needed, initialization of that data during subprogram elaboration is no more or less efficient than initialization of the same data during subprogram execution.
- Aggregate assignment is just as efficient, if not slightly more efficient, as individual component assignment.
- Initialization of package state data during elaboration is more efficient than initialization of the same state data through the use of a subprogram call.

  [NOTE: This efficiency is relevant only when short initialization time is a requirement and the amount of CPU time needed to perform initialization is small relative to the CPU time needed to perform a procedure call.]
- Defining records with default values does not incur any run-time overhead when those defaults are overridden during object declaration. However, accepting the default values during declaration and unconditionally initializing the same object during subprogram execution will generate twice as many assignments.
- Finally, data elaborated and initialized with a subprogram body should be minimized while retaining information-hiding and encapsulation principles.

3.1.4 Generic Units

3.1.4.1 PURPOSE

To increase the level of software reuse in flight dynamics, simulators have used Ada generic units. These generic units have saved a significant amount of effort in

subsequent simulator projects (Reference 4). However, if a nongeneric unit is significantly more efficient than a generic unit, the generic unit may not be reused. This test examines the CPU and memory overhead associated with the use of a generic unit for matrix multiplication.

3.1.4.2 METHOD

This test measures the CPU and memory utilization of multiplying two 3x3 and two 6x6 matrixes. This test consists of two experiments:

- Multiplying two 3x3 matrixes using a generic unit and a nongeneric unit
- Multiplying two 6x6 matrixes using a generic unit and a nongeneric unit

The matrix types were constrained two-dimensional arrays of single-precision floating-point numbers. Figure 3-5 shows the generic and nongeneric source code solution for this test.

3.1.4.3 RESULTS

As Table 3-5 and Figure 3-6 show, no measurable difference exists in processing times using the generic and nongeneric 3x3 matrix units, and very little difference exists in processing times of the 6x6 matrix units. Additionally, the differences in object code sizes are insignificant.

3.1.4.4 ANALYSIS

A compiler may implement generic units using an inline expansion algorithm or a code-sharing algorithm. The inline expansion algorithm does just as the name suggests: each instantiation of a generic unit is expanded inline at the point of instantiation. The code-sharing algorithm generates one object code template and passes the actual generic parameters for each instantiation at run time.

The advantage of the inline expansion algorithm is that each instantiation of the generic unit may be optimized on the basis of actual parameters passed in during instantiation. This effect may be exploited, as shown in Section 3.1.5, "Conditional Compilation." The advantage of the shared-code algorithm is that each instantiation does not generate an additional object code copy. Instantiations may be nested within the scope of the type visibility without increasing object code size. This feature encourages strong typing. This is particularly important for targeted hardware with limited memory. However, code sharing may be important on virtual machines to minimize the number of page faults.

DEC Ada version 1.5 uses an inline expansion algorithm, which test results confirm. The instantiations of the generic units are as efficient, both in terms of CPU time and memory, as the corresponding nongeneric units. However, if multiple instantiations are made, the memory utilization will increase linearly. A shared code algorithm will

Generic Solution	Nongeneric Solution

generic
  type REAL is <>;
  type INDEX is range <>;
  type MATRIX is
  array (INDEX,INDEX) of REAL;
package Generic_Matrix is
  function "*"
  (M1, M2 : in MATRIX)
  return MATRIX;
end Generic_Matrix;
 	
package body Generic_Matrix is
 	
  function "*"
  (M1, M2 : in MATRIX)
  return MATRIX is
  R : MATRIX;
  begin
  ...
  return R;
  end "*";
 	
end Generic_Matrix;
 	
package Matrix_Types is
  type REAL6 is digits 6;
  type INDEX3 is range 1..3;
  type MATRIX33 is
  array (INDEX3,INDEX3) of REAL6;
  type INDEX6 is range 1..6;
  type MATRIX66 is
  array (INDEX6,INDEX6) of REAL6;
end Matrix_Types;
 	
with Matrix_Types;
use Matrix_Types;
package Matrix_3D is new
  Generic_Matrix
  (REAL => REAL6,
  INDEX => INDEX3,
  MATRIX => MATRIX33);
with Matrix_Types;
use Matrix_Types;
package Matrix_6D is new
  Generic_Matrix
  (REAL => REAL6,
  INDEX => INDEX6,
  MATRIX => MATRIX66);
	package Matrix3 is
	
	  type REAL6 is digits 6;
	  type INDEX3 is range 1..3;
	  type MATRIX33 is
	  array (INDEX3,INDEX3) of REAL6;
	
	  function "*"
	  (M1, M2 : in MATRIX33)
	  return MATRIX33;
	
	end Matrix3;
	
	package body Matrix3 is
	
	  function "*"
	  (M1, M2 : in MATRIX33)
	  return MATRIX33 is
	  R : MATRIX33;
	  begin
	  ...
	  return R;
	  end "*";
	
	end Matrix3;
	
	package Matrix6 is
	
	  type REAL6 is digits 6;
	  type INDEX6 is range 1..6;
	  type MATRIX66 is
	  array (INDEX6,INDEX6) of REAL6;
	
	  function "*"
	  (M1, M2 : in MATRIX66)
	  return MATRIX66;
	
	end Matrix6;
	
	package body Matrix6 is
	
	  function "*"
	  (M1, M2 : in MATRIX66)
	  return MATRIX66 is
	  R : MATRIX66;
	  begin
	  ...
	  return R;
	  end "*";
	
	end Matrix6;

6344G(30)-27

Figure 3-5. A Generic Solution and a Nongeneric Solution

Table 3-5. CPU Time and Memory Usage for Generic Units

Experiment	CPU Time (µs)	Object Code Size (bytes)
3x3 Generic Unit	117.5	190
3x3 Nongeneric Unit	117.5	187
6x6 Generic Unit	768.8	197
6x6 Nongeneric Unit	737.5	196

6344G(30)-09

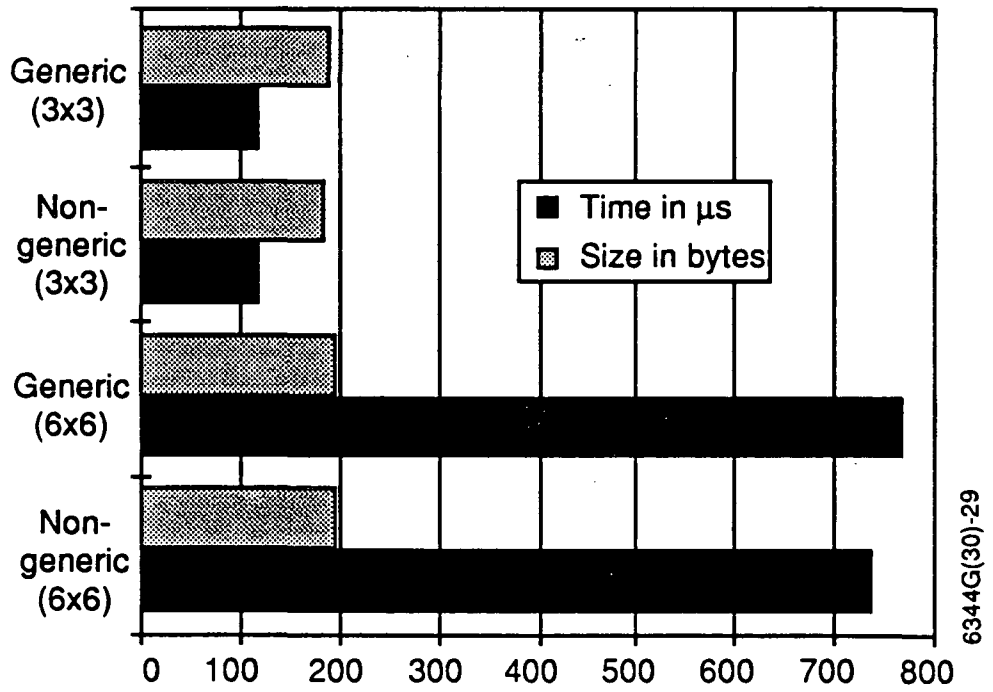

Figure 3-6. Comparison of Times and Sizes for Generic Units

be necessary to exploit strong typing and encourage abstraction and information hiding without adversely affecting performance (References 8, 9, and 10).

[NOTE: DEC Ada 2.0 supports the code-sharing algorithm for certain types of generic units (Reference 11).]

The implications of this test are as follows:

- Use a compiler that supports both inline expansion and code sharing of generics.
- Use an inline expansion algorithm when few instantiations of a generic unit are necessary or when a particular instantiation must be optimized at compile time.
- Use a code-sharing algorithm when multiple instantiations of a generic unit are necessary or when memory space must be optimized at compile time.

3.1.5 Conditional Compilation

3.1.5.1 PURPOSE

Section 10.6 (Program Optimization) of the Ada LRM (Reference 1) states:

> A compiler may find that some statements or subprograms will never be executed, for example, if their execution depends on a condition known to be FALSE. The corresponding object machine code can then be omitted. This rule permits the effect of conditional compilation within the language.

This optimization technique has applications throughout our software systems in flight dynamics. A Boolean flag may be included with a generic unit that controls the accuracy of the resulting object code. For example, consider the following generic unit.

```
generic
 Correct_For_Aberration_Effects : in BOOLEAN := FALSE;
package Moon is
 function Position (At_Time : in TIME_TYPE)
    return GCI_VECTOR;
 ...
end Moon;
```

In the body of the function Moon.Position, there is a block of code resembling the following:

```
if Correct_For_Aberration_Effects then
 ... --code to do correction goes in here
end if;
```

For a dynamics simulator or for a mission in which the Moon's interference with sensors is relevant, the generic might be instantiated with the correction flag set to TRUE to achieve the greatest accuracy. However, for most other applications, the generic would probably be instantiated with the correction set to FALSE. This would result in less accurate, yet more efficient, executable code because the "if" statement could be completely optimized away.

3.1.5.2 METHOD

Current flight dynamics simulators check a run-time variable to determine whether the user selected debug output. This test addressed the run-time overhead of having an infrequently used feature in a delivered system. The alternative examined was the use of a compilation time constant.

The test involved the debug output capability that is included in all of the simulators developed in the FDD. The package Debug_Collector was originally developed on the GRODY project. The package's capabilities were expanded on the GOADA project to handle two floating-point types (single and double precision). The package was modified to be generic on the Upper Atmosphere Research Satellite (UARS) telemetry simulator (UARSTELS) project. Finally, the package was optimized on the EUVETELS project.

The optimizations incorporated by the EUVETELS project included the use of **pragma** INLINE for functions that other components in the system use to determine whether the user has enabled debug output. In addition, a Boolean formal parameter (Debug On) was added to the generic parameter list.

The test executed by the Ada performance study compared the cost of checking a run-time variable against the cost of checking a compilation time constant. In both cases, the variable/constant is set to FALSE to determine the run-time overhead associated with having an infrequently used capability built into a system. This overhead includes the CPU time and the amount of executable code generated. To achieve this, two instantiations of the Debug_Collector package were used: the first sets the Debug_On constant to TRUE, and the second sets the constant to FALSE.

3.1.5.3 RESULTS

As Table 3-6 shows, the CPU cost of checking a run-time variable that is set to FALSE is approximately 1 ms, while the run-time CPU cost of checking a compilation time constant that is set to FALSE is zero. This result suggests that the Ada compiler has optimized away the "if" statement completely. Inspection of the assembly code confirmed this fact. The size of the executable code was also reduced due to this elimination.

Table 3-6. Conditional Compilation Results

Options	CPU to Check (μs)
Run-Time Variable	0.9
Compilation-Time Constant	0.0

6344G(30)-10

3.1.5.4 ANALYSIS

The implications of this test exceed this simple debug example. One of the key points demonstrated by this test was that the generality (and in the case of Ada, "genericness") of the software was not sacrificed to gain a more CPU- and memory-efficient system. This test has important implications for designers of reusable Ada components:

- Proper design for reusable, generic components will provide sufficient flexibility for designers of future missions to choose between accuracy and efficiency without modifying the component.

3.1.6 Object-Oriented Programming

3.1.6.1 PURPOSE

Two of the fundamental principles of OOP are polymorphism and inheritance. Ada does not directly support these principles. However, the designer may simulate the effect of inheritance and polymorphism through the use of variant records and enumeration types. These OOP principles, whether direct or indirect, incur a certain amount of run-time overhead and were intentionally excluded from the Ada language because of potential performance problems.

3.1.6.2 METHOD

The first test method (inheritance method) passes specific sensor data into a subprogram to process sensor data. The subprogram then uses a case statement to determine which type of sensor data is being processed and calls the appropriate data processing routine. The second test method calls the appropriate data processing routine directly. The two test procedures and the Process Sensor Data subprogram are shown in Figure 3-7, along with the data structures used.

3.1.6.3 RESULTS

Results for the two tests are summarized in Table 3-7.

3.1.6.4 ANALYSIS

As expected, calling the subprogram to process the sensor data directly takes less time than calling the intermediate subprogram to make the run-time decision about which specific routine to call. Using the intermediate subprogram takes approximately 6 ms longer than calling the desired subprogram directly. This extra expense may be considered small if the subprogram called consumes a large amount of CPU time (e.g., 1000 ms). However, if this approach is a fundamental design decision and several layers of a class hierarchy are created in this fashion, nearly all subprogram calls could incur this performance penalty. The implication of this test is as follows:

- When using the OOP principles of inheritance and polymorphism, minimize the class hierarchy and maximize the processing performed by the subprogram called.

```
type KIND is (CSS, FSS, FHST);

type FSS_Type is record
   Alpha Beta: FLOAT;
end record;

type CSS_Type is record
   Y, Z: FLOAT;
end record;

type FHST_Type is record
   Theta, Phi: FLOAT;
end record;

type SENSOR (K: KIND) is record
   case K is
      when CSS => CSS_Stuff: CSS_Type
      when FSS => FSS_Stuff: FSS_Type
      when FHST => FHST_Stuff: FHST_Type;
   end case;
end record;

FSS1: SENSOR(FSS);
X: INTEGER; -- control variable

procedure Process_Data (For_Sensor : In SENSOR) is
begin
   case For_Sensor.K is
      when CSS => Process_CSS_Data;
      when FSS => Process_FSS_Data;
      when FHST => Process_FHST_Data;
   end case;
end Process_Data;

procedure Test-Tag is
begin
   X: =4; -- control statement
   Process_Data (FSS1);
end Test_Tag;

procedure Test_Direct is
begin
   X: =4; -- control statement
   Process_FSS_Data;
end Test_Direct
```

6344G(30)-11

Figure 3-7. Method for Simulating Inheritance and Polymorphism

Table 3-7. Measuring the Overhead Associated With OOP

Direct Method (μs)	OOP Method (μs)
3.2	9.2

6344G(30)-12

3.2 IMPLEMENTATION-ORIENTED TESTS

3.2.1 Matrix Storage

3.2.1.1 PURPOSE

Flight dynamics simulators use two-dimensional arrays in quantity. In some cases, time could be saved by using the most efficient access method to these arrays. The matrix storage test investigated the cost of the different addressing mechanisms of a matrix.

3.2.1.2 METHOD

The test involved using a constrained integer matrix. The operations on the matrix consisted of an assignment of a calculated value to the matrix cell. Three methods were used to address the matrix indexes:

- Row-major order of processing
- Column-major order of processing
- A series of statements addressing each cell individually

Figure 3-8 shows the row- and column-major code fragments used for this test. The third approach used a series nine of statements that replace the double for-loops.

Row-Major Addressing	Column-Major Addressing

for i in 1..3 loop --i
  for j in 1..3 loop --j
  Matrix(i, j) := <expression>;
  end loop;
end loop;
	for j in 1..3 loop --j
	  for i in 1..3 loop --i
	  Matrix(i, j) := <expression>;
	  end loop;
	end loop;

6344G(30)-13

Figure 3-8. Sample Code Segment From the Matrix Storage Test

3.2.1.3 RESULTS

The row-major order method of processing was the standard by which the test was measured. This is the DEC Ada storage and addressing method when processing a matrix. As shown in Figure 3-9 and Table 3-8, the column-major order method of processing was nearly two times slower than the row-major order method. The fastest method of access was to assign each matrix cell individually.

3.2.1.4 ANALYSIS

The reason for the higher cost of column-major processing was the additional memory management necessary to move the assigned value into each cell. As mentioned

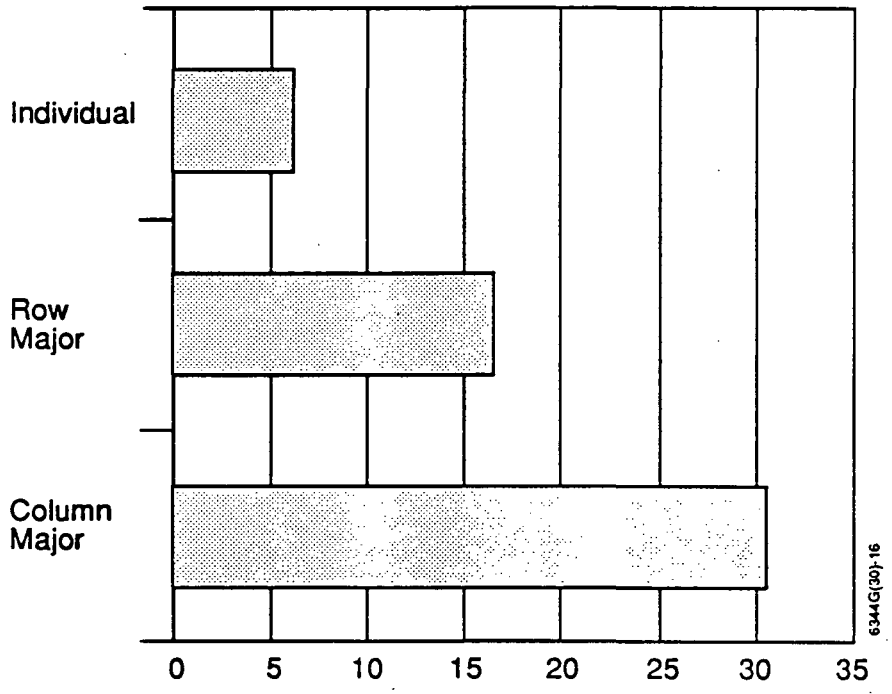

Figure 3-9. Comparison of Times (μs) for Matrix Storage

Table 3-8. CPU Time (μs) Results of the Matrix Storage Test

Access Method	CPU Time (μs)
Row-major processing	17.2
Column-major processing	30.5
Series of individual statements	5.9

6344G(30)-15

previously, the DEC Ada method for storing matrixes in memory is row major. This makes row-major addressing a simple operation of addition to the current memory location to get to the next memory location. When accessing a matrix in a column-major mode, the address calculation is more complex and requires more CPU time to complete.

When using a series of individual statements to access the matrix, the CPU time was significantly reduced. This created more source and object code but allowed the compiler to optimize away the matrix address calculations. Instead of referencing a matrix by two variable indexes, the code was written to access a literal index [e.g., matrix(1,1)]. Because the matrix type was constrained, the code was checked at

compile time for any range or access errors, and therefore the run-time version did not include these checks. The implications of this test follow:

- For most Ada implementations (in this case, DEC Ada), the recommended method of matrix processing is row major.
- It is more time efficient to reference each element individually. For small arrays, this method would only slightly increase the size of the executable code.

3.2.2 Logical Operators

3.2.2.1 PURPOSE

The logical operators test group compares different methods of implementing compound logical expressions. Comparisons are made of tests with and without short circuit operators and with nested "if" statements, where applicable. The same tests were run with compound expressions of two variables, two external function calls, and a variable and an external function call, resulting in a total of three experiments with multiple tests each.

3.2.2.2 METHOD

Each test in this experiment was designed and developed using the PIWG methodology. There are five test procedures ("and," "and then," "or," "or else," nested "if"), each containing two input Boolean parameters, *a* and *b*, which are used in the compound logical expression. Each test was run twice, first with the first input parameter (a) equal to TRUE and second with the first input parameter (a) equal to FALSE. This input parameter (a) is used in the first half of the compound logical expression and determines whether short circuiting is possible. For the "and," "and then," and nested "if" tests, the second input parameter (b) defaults to FALSE. For the "or" and "or else" tests, the second input parameter (b) defaults to TRUE. The tests are organized into three experiments as described below.

METHOD FOR a < boolean operator > b TESTS

The first set of tests compares expressions of the form

a and *b*

a and then *b*

a or *b*

a or else *b*

where *a* and *b* are parameters input to the test procedures. In addition, a nested "if" test of the form "if *a* then if *b*" was run to compare the results to the "and" and "and

then" results. Two sets of tests were run, one with *a* set to TRUE and one with *a* set to FALSE.

METHOD FOR f(a) < boolean expression > (b) TESTS

The second set of tests compares expressions of the form

f(*a*) and *f*(*b*)

f(*a*) and then *f*(*b*)

f(*a*) or *f*(*b*)

f(*a*) or else *f*(*b*)

where *a* and *b* are parameters input to the test procedures and the external function *f* returns whatever Boolean value its argument possesses. Two sets of tests were run, one with *a* set to TRUE and one with *a* set to FALSE.

METHOD FOR a < boolean expression > f(b) TESTS

The third set of tests compares expressions of the form

a and *f*(*b*)

a and then *f*(*b*)

a or *f*(*b*)

a or else *f*(*b*)

where *a* and *b* are parameters input to the test procedures and the external function *f* returns whatever Boolean value its argument possesses. Two sets of tests were run, one with *a* set to TRUE and one with *a* set to FALSE.

3.2.2.3 RESULTS

Results from each of the experiments are presented in Tables 3-9, 3-10, and 3-11. The nested "if" test was not replicated for the "*f*(*a*) < boolean expression > *f*(*b*)" tests or the "*a* < boolean expression > *f*(*b*)" tests because of its similarity to the "and then" short circuit form demonstrated in Table 3-9.

3.2.2.4 ANALYSIS

ANALYSIS OF a < boolean operator > b TESTS

Examination of the assembly code for the tests with and without short circuit operators shows that the compiler generates the same code for both sets of tests; that is, short circuiting is performed in all cases. Because of this, the CPU times for the "and" and

Table 3-9. Test Results of *a* < boolean operator > *b*

A	B	Operator	CPU time (μs)
TRUE	FALSE	and	1.6
TRUE	FALSE	and then	1.5
TRUE	FALSE	if A then if B	1.5
FALSE	FALSE	and	0.9
FALSE	FALSE	and then	1.1
FALSE	FALSE	if A then if B	1.2
TRUE	TRUE	or	0.5
TRUE	TRUE	or else	0.4
FALSE	TRUE	or	0.5
FALSE	TRUE	or else	0.5

6344G(30)-17

Table 3-10. Test Results of *f(a)* < boolean expression > *f(b)*

F(A)	F(B)	Operator	CPU time (μs)
TRUE	FALSE	and	15.2
TRUE	FALSE	and then	15.2
FALSE	FALSE	and	15.2
FALSE	FALSE	and then	8.0
TRUE	TRUE	or	15.0
TRUE	TRUE	or else	8.1
FALSE	TRUE	or	14.9
FALSE	TRUE	or else	15.0

6344G(30)-18

Table 3-11. Test Results of a < boolean expression > *f(b)*

A	F(B)	Operator	CPU time (μs)
TRUE	FALSE	and	8.0
TRUE	FALSE	and then	8.3
FALSE	FALSE	and	7.9
FALSE	FALSE	and then	0.9
TRUE	TRUE	or	7.3
TRUE	TRUE	or else	0.7
FALSE	TRUE	or	7.5
FALSE	TRUE	or else	8.0

6344G(30)-19

"and then" tests were effectively the same (negligible differences due to the effects of a nonstandalone machine), and the CPU times for the "or" and "or else" tests were effectively the same (negligible differences due to the effects of a nonstandalone machine).

ANALYSIS OF *f(a)* < boolean expression > *f(b)* TESTS

Examination of the assembly code showed that the tests act as expected; that is, short circuiting is only performed for the "and then" and "or else" tests. Therefore, in cases where both expressions do not have to be evaluated because *f(a)* is FALSE in the "and then" case or *f(a)* is TRUE in the "or else" case, the short circuit tests are faster. In the other cases, the CPU times are effectively the same (differences due to the effects of a nonstandalone machine).

ANALYSIS OF *a* > boolean expression > *f(b)* TESTS

Examination of the assembly code for the "and" and "or" tests shows that the compiler evaluates the function call first and then performs short circuiting logic. To allow for the chance that the function may change the variable *a*, the contents of *a* are moved to a different location before the function call. If *a* must be evaluated, the contents of this new location are evaluated, not the contents of the original location for *a*.

Because *f(b)* is always FALSE for the "and" tests and *f(b)* is always TRUE for the "or" tests, these tests only require evaluation of *f(b)* for reasons described in the preceding paragraph. It should be noted that small differences in the numbers [e.g., 8.0 for the first test to evaluate *f(b)* and 7.9 for the third test for the same task] should be treated as

system "noise" (the effects of a nonstandalone machine) and that the numbers should be treated as effectively equal.

3.2.2.5 SUMMARY ANALYSIS

The DEC Ada compiler will short circuit a Boolean expression whenever possible. This possibility occurs when at least one of the Boolean operands is a variable. This automatic short circuiting is not possible when both operands are function calls. In this case, the function calls may have desired side effects and both must be executed unless explicit short circuiting is specified. The implication of the logical operators test group is as follows:

- Short-circuit Boolean forms **should** be used to act as a guard for the second expression (Reference 7). Additionally, short circuit Boolean forms **may** be used to explicitly short circuit an expression containing function calls for performance-critical applications.

3.2.3 pragma INLINE

3.2.3.1 PURPOSE

Flight dynamics simulators contain a large number of procedure and function calls to simple call-throughs and selectors. The overhead of making these calls can slow the performance of any simulator. This test measured the use of **pragma** INLINE as an alternative to calling a routine.

3.2.3.2 METHOD

A two-step method was studied in this experiment. First two routines were written and compiled without any pragma. Those routines were then copied and **pragma** INLINE was added to the code of each procedure. The two test cases included a procedure to multiply a number and a procedure that did nothing (a stub). The following experiments make up the test:

- Make an external procedure call to do an operation
- Make an external stub call to do nothing
- Make the same call as in test 1 but use **pragma** INLINE
- Make the same call as in test 2 but use **pragma** INLINE

The results should indicate that the use of **pragma** INLINE improves performance while sacrificing compiled code size. The **pragma** INLINE directive will bring in the external reference and do the operations of the procedure in the body of the calling routine. The bottom line is that this may be a significant performance enhancement to Ada while allowing the designer to retain the modularity of the design.

3.2.3.3 RESULTS

The comparisons show that the calls to the routines compiled with **pragma** INLINE were much faster than those compiled without. As Table 3-12 shows, using this compiler directive improved the performance of the test routines by more than ten-fold.

Table 3-12. Measuring the Effect of pragma INLINE

Method	CPU time (μs)
Call a procedure to do an operation	59.0
Call a STUB to do nothing	45.9
Call a procedure to do an operation (INLINE)	6.3
Call a STUB to do nothing (INLINE)	0.4

6344G(30)-20

3.2.3.4 ANALYSIS

Pragma INLINE allows the compiler to build the code of the external procedure into the main calling body. Because no overhead in stack maintenance and address change exists for the subroutine call, **pragma** INLINE improves performance. If parameters are passed to the called routine, performance will improve even more. If a procedure is called from only one location, the final code size may not increase at all. In fact, due to the elimination of stack maintenance and address changing, the executable size may decrease.

A compilation issue must be understood when using the **pragma** INLINE. The compile time cost can be much higher when the inlined subprogram is changed. This change requires all the units that call the subprogram to be recompiled so that the new or modified object code can be "re-inlined." The solution is to leave out the **pragma** INLINE directive until the procedure has reached its final state. At this time, the directive can be added with the effect of recompiling only once.

The implication for the use of **pragma** INLINE is as follows:

- Use **pragma** INLINE when the subprogram is small and its expected use is to be referenced in few locations and be called several times dynamically.

3.2.4 String-to-Enumeration Type Conversion

3.2.4.1 PURPOSE

Current flight dynamics simulators contain a central logical data base. The physical data are distributed throughout the simulators in the appropriate packages. The

logical data base provides keys (strings) that map into the physical data. The logical data base converts these strings to the appropriate enumeration type to retrieve the corresponding data. This test assesses the performance implications of this approach.

3.2.4.2 METHOD

The PCA analysis indicated that the string manipulation and conversion to an enumeration type was consuming 14.5 percent of GOADA's CPU time. In addition, the several calls to the logical data base package were suspected of consuming a significant amount of CPU time. References to the logical data base package were modified to call the appropriate package directly, rather than using the logical data base. The differences between the two methods are shown using PDL in Figure 3-10. The main difference between the two approaches is the time in which certain decisions are made. In the logical data base approach, the decision of what package to call and what enumeration type to use is made at run time. In the direct call approach, these decisions are made at compile time.

Logical Data Base Approach	Direct Call Approach
1. Call logical data base ("Prefix_Identifier") 2. Extract the Prefix 3. Determine package from the Prefix 4. Convert the keys to the appropriate enumeration type 5. Call the appropriate package (Enumeration_Literal)	1. Call the appropriate package (Enumeration_Literal)

6344G(30)-22

Figure 3-10. PDL Comparison of Logical Data Base and Direct Call Approaches

3.2.4.3 RESULTS

As shown in Table 3-13, the change to direct referencing improved GOADA's performance by 70 seconds, or 17.3 percent. The use of the logical data base was restricted to the user interface.

[NOTE: A similar performance improvement was observed during the performance tuning phase of the EUVETELS software development project (Reference 4).]

Table 3-13. Results From Logical Data Base Versus Direct Call

Time	Baseline Version of GOADA	Direct Call Approach
Total CPU Time	6:45	5:35

6344G(30)-23

3.2.4.4 ANALYSIS

The conversion from string to an enumeration literal is CPU intensive compared with the numerical operations typically performed in a dynamics simulator. The implication of this test is as follows:

- Restrict string-to-enumeration type conversions to non-CPU critical segments of the system. Statically binding to a package and type, rather than dynamically converting and testing, is the preferred method within performance-critical loops (e.g., simulation loop).

SECTION 4 – CONCLUSIONS

The Ada performance study had three objectives: (1) to determine which design and implementation alternatives lead to accelerated run times; (2) to determine what, if any, trade-offs are made by choosing these alternatives; and (3) to develop guidelines to aid future Ada development efforts in the FDD. Section 4.1 summarizes the lessons learned about the compiler and current design and implementation approaches. Section 4.2 summarizes the results of each of the performance tests described in Section 3 and provides recommendations for future development projects.

4.1 LESSONS LEARNED

A valuable side effect of this performance study is a much better understanding of both the software development environment and the current design and implementation approaches of FDD simulators. Section 4.1.1 summarizes the lessons learned about the DEC Ada 1.5-44 version of the compiler. Section 4.1.2 summarizes the lessons learned about current simulator development.

4.1.1 Compiler Options and Problems

The DEC Ada 1.5-44 compiler, Ada Compilation System (ACS), and the VAX software engineering tools (VAXset) represent one of the more mature Ada development environments available. Many optimization features were discovered by reviewing the intermediate assembler code generated by the compiler. However, as with any large software system, some known limitations and bugs exist.

The following optimizations performed by the compiler were discovered during the course of this study:

- Array assignment is implemented as a single "block move" command rather than individual component moves (see test group 1, scheduling).
- Using **pragma** SUPPRESS_ALL will not only eliminate the run-time constraint checks but will eliminate all constraint checking and exception handling object code (see test group 2, unconstrained structures).
- Although subprogram elaboration code might be a large portion of the system, the CPU consumption is minimal (see test group 3, initialization and elaboration).
- The compiler recognizes and eliminates unreachable code even when a combination of **pragma** INLINE and generics is used (see test group 5, conditional compilation).

- Compound Boolean expressions are automatically short circuited whenever possible (see test group 7, logical operators).

The following limitations were discovered or considered to be significant to Ada development:

- Using a nested-loop algorithm with unconstrained arrays is inefficiently implemented with significant loop-invariant object code (see test group 2, unconstrained structures).
- Local composite objects that are not referenced, either through accident or intent, are not optimized away (see test group 3, initialization and elaboration).
- Code-sharing among similar instantiations of generic units is not possible (see test group 4, generic units).
- The PCA does not display the CPU time profiles of nested units and overloaded operators in a straightforward manner.

The latest version of the DEC Ada compiler (2.0-18) seems to correct some of these problems. Further analysis of this new compiler release is necessary before any final determination can be made.

4.1.2 Estimating Simulator Performance

As a result of this performance study, more accurate estimation of run-time performance for future FDD simulators is possible. Assuming future dynamics simulators are similar in function to GOADA, a more accurate performance estimation is possible given the following information:

- The run-time performance for a typical run of the GOADA simulator is 6 minutes, 45 seconds for a 20-minute simulation. This yields a 1:3 simulation time-to-real time ratio.
- The performance profile generated by PCA of a typical GOADA run (see Figure A-2) shows the distribution of the CPU run-time resource throughout the simulator.
- The results of this study suggest more optimal design and implementation alternatives.

Table 4-1 combines the results of the Ada Performance Study with the PCA performance profile of GOADA.

The first row of Table 4-1 shows the performance difference between the baseline scheduler in GOADA and the looping scheduler alternative (see test group 1, scheduling).

Table 4-1. Impact of Measured Performance Results on Dynamics Simulators

Alternative	GOADA	Study Results	Difference
1. Looping scheduler	10.7%	2.2%	8.5%
2. Bypass logical data base	14.5%	1.8%	12.7%
3. Conditional compile debug code	2.1%	0.0%	2.1%
4. Use static data structures	45.0%	13.0%	32.0%
5. Optimized utility packages	26.6%	5.3%	21.3%
Total Percentages	98.9%	22.3%	76.6%

6344G(30)-24

Another option is to use the hard-coded approach for the scheduler. However, the hard-coded approach sacrifices all flexibility in the interest of performance. For this reason, the more flexible looping approach is the recommended alternative.

The second row highlights the difference between accessing the logical data base and accessing the physical data directly (see test group 10, string-to-enumeration conversion). The third row recommends the conditionally compiled debug code (see test group 5, conditional compilation). The fourth row is the estimated result of using a static record structure instead of a dynamic structure in all simulator packages (see test group 2, unconstrained structures).

The fifth row is based on the result of comparing GOADA's baseline matrix multiply function with the optimized matrix multiply function (Reference 12). Because the FDD deals with mainly three dimensions, an optimized set of utilities can be developed on that basis. The fully optimized version required less than one-fifth of the CPU time required for the baseline version.

As Table 4-1 shows, a dynamics simulator that is similar to GOADA and is implemented with the results of this study would consume 76.6 percent less CPU than the current version, more than quadrupling the speed. This would yield an upper bound estimate of 95 seconds to perform a 20-minute simulation run, or approximately a 1:13 simulation time-to-real time ratio.

This estimate is an upper bound for three reasons. First, this study examined a representative, rather than exhaustive, list of design and implementation alternatives; that is, only those alternatives that held the most promise of a large performance difference were studied. Many other alternatives may be available that offer only minor gains. However, the combined performance gain of all alternatives may be significant.

Second, coding optimizations to GOADA, or any simulator, were not studied. The goal of this study was to identify which design and implementation alternatives would lead to optimal systems. Line-by-line micro-optimizations on a simulator only provide information on final efficiency and lack the needed information on how to systematically predict and achieve that level of efficiency.

Finally, the DEC Ada 1.5-44 compiler is a relatively error-free first attempt at an Ada compilation system. The next generation of Ada compilers, which include DEC Ada 2.0, is now available on the market. These second-generation compilation systems represent improvements in the user (programmer) interface, including efficiency improvements.

4.2 RECOMMENDATIONS

This section summarizes the results from the performance tests described, in detail, in Section 3. The recommendations for developers of future FDD Ada systems are separated by life-cycle phase with potential trade-offs and the associated test group(s) identified.

4.2.1 Requirements Analysis

One recommendation affects the requirements analysis and early design phases of a project.

R-1. Match the data in the problem space (flight dynamics) to the appropriate data structure in the solution space.

Trade-offs: None

Reference: Test group 1, scheduling

4.2.2 Design

Five recommendations affect the design phase of a project.

R-2. Match the algorithm to both the data structure and the data.

Trade-offs: None

Reference: Test group 1, scheduling, and test group 2, unconstrained structures.

R-3. Design procedures and functions for each package that map data of a general type to the data (hidden) optimal type.

Trade-offs: More lines of code

Reference: Test group 2, unconstrained structures

R-5. Design generic components to allow users to choose between accuracy and efficiency.

Trade-offs: Higher complexity, more lines of code

Reference: Test group 5, conditional compilation

R-6. Performance-critical loops should not include any string-to-enumeration conversions.

Trade-offs: Less generality, more lines of code

Reference: Test group 10, string-to-enumeration conversion

4.2.3 Implementation

Three recommendations affect the implementation phase of a project.

R-7. Looping structures should access arrays in row-major order.

Trade-offs: None

Reference: Test group 7, matrix storage

R-8. Use attributes wherever possible when unconstrained structures are necessary.

Trade-offs: None

Reference: Test group 2, unconstrained structures

R-9. Only use short-circuit control forms for performance reasons when the expression contains function calls that have no side effects.

Trade-offs: None

Reference: Test group 6, logical operators

4.2.4 Maintenance

One recommendation affects the maintenance phase of a project.

R-10. Modifications must address both the algorithmic and the data structure changes to ensure that they both still match the problem.

Trade-offs: None

Reference: Test group 1, scheduling

APPENDIX A—APPROACH TO MEASUREMENT

To measure the run-time performance of design alternatives and language features, two fundamental approaches were used. The first approach measured the run-time improvement of existing systems after an alternative had been incorporated into a baseline version of the system. The second approach used the ACM SIGADA PIWG test suite and added tests specific to the flight dynamics environment.

A.1 OVERVIEW

Benchmark programs are commonly used to evaluate the performance of design alternatives and language features. Such benchmark programs include: (1) sample applications such as sorting programs or, as in the FDD, simulators; (2) programs to measure the overhead associated with a design alternative or language feature; and (3) synthetic benchmarks designed to measure the time needed to execute a representative mix of statements (e.g., Whetstone, Dhrystone) (Reference 13). The first approach used by this study falls into the first benchmark category, and the second approach falls into the last two.

To measure the overhead of a design alternative or language feature, the dual-loop approach is used to subtract the overhead associated with control statements that aid in performing the measurement. This approach utilizes a control loop and a test loop; the test loop contains everything contained in the control loop, plus the alternative being measured. This is described further in Section A.3. A major factor in designing a dual-loop benchmark is compiler optimization. It is critical that the code generated by the compiler for both loops be identical except for the quantity being measured (Reference 14). In addition, it is necessary to ensure that the statement or sequence of statements being tested does not get optimized away.

Although the dual-loop approach can be used for synthetic benchmarks and applications, this technique is not required if the running time of the program is long in comparison to the system clock resolution (Reference 14). Instead, the CPU time can be sampled at the beginning of the program and again after a number of iterations of the program. The time for the benchmark/application is then (CPU_Stop minus CPU_Start)/Number_Iterations. The same measurement can be achieved by submitting the test program to run as a batch job and obtaining the CPU time from the batch log file. This CPU time can then be divided by the number of times the sequence of statements being measured is executed in the main control loop of the test program.

It is important to understand the run-time environment in which the benchmarks are run when interpreting test results. VMS checks the timer queues once per second, which can affect measurement accuracy. Under VMS, the Ada run-time system is bundled with the release of the operating system and is installed as a shareable executable image. Consequently, DEC Ada performance is directly dependent on the

installed version of VMS. There is also a degree of uncertainty present when using CPU timers provided in time-shared systems like VMS. In the presence of other jobs, CPU timers charge ticks to the running process when the wall clock is updated. It is therefore possible for time to be charged to active processes inaccurately, since context switches can occur at any time. Finally, it cannot be assumed that running benchmarks for a hosted system in batch during low utilization (e.g., at 11 p.m.) guarantees standalone conditions (References 14 and 15). Therefore, the benchmarks to test individual design alternatives were run on the weekend to minimize these effects.

A.2 FIRST APPROACH – SIMULATOR

Several of the design alternatives examined by this study were tested and analyzed in the context of an FDD simulator, GOADA. Alternatives were chosen to be implemented in the context of this simulator for the following reasons:

1. They were simulator-specific (e.g., different ways of implementing the scheduler).

2. They could be implemented in an isolated part of the simulator where their impact could easily be measured using the VAX PCA.

3. They could be implemented in an isolated part of the simulator and still have a measurable effect on the time required for a 20-minute simulation run.

The baselined version of GOADA was used to test each of the design alternatives. CPU times were obtained for 20-minute simulation runs of the baselined version from the log files created by batch runs. PCA was used to obtain a profile of the simulator. This profile showed what percentage of the CPU time was spent in each Ada package of the simulator. The VAX manual *Guide to VAX Performance and Coverage Analyzer* (Reference 16) contains more information on PCA.

Design alternatives were incorporated into the baselined version of the simulator. New CPU times were obtained for 20-minute simulation runs from the log files created by batch runs and new profiles obtained using PCA. Figure A-1 shows the accounting information contained in a batch log file, and Figure A-2 shows sample PCA output. From these two pieces of information, the impact of each design alternative was assessed.

Accounting information

Buffered I/O count:	109	Peak working set size:	4096
Direct I/O count:	1132	Peak page file size:	15304
Page faults:	11766	Mounted volumes:	0
Charged CPU time:	0 00:06:45.08	Elapsed time:	0 00:09:02.47

6344G(30)-25

Figure A-1. Sample Batch Log File Accounting Information

```
                    VAX Performance and Coverage Analyzer
                CPU Sampling Data (11219 data points total) - "*"

Bucket Name          +----+-------------------------------------------------+
PROGRAM_ADDRESS\     |
 UTILITIES_ . . . . .|************************************************** 23.2%
 SIMULATION_SCHEDULE|
  R . . . . . .      |**********************                             10.7%
 SEARCH_STRING  . .  |****************                                    7.5%
 SPACECRAFT_ATTITUDE|****************                                    7.5%
 DATABASE_MANAGER .  |***************                                     7.0%
 ADDING_UTILITIES .  |**********                                         4.7%
 EARTH_SENSOR . . .  |********                                           3.7%
 UTILITIES_LONG_  .  |******                                             3.0%
 DATABASE_TYPES_  .  |******                                             2.9%
 SPACECRAFT_WHEELS   |******                                             2.9%
 AOCS_PROCESSOR . .  |******                                             2.6%
 SPACECRAFT_EPHEMERI|
  S . . . . . .      |*****                                              2.5%
 ENVIRONMENTAL_TORQU|
  ES  . . . . . .    |*****                                              2.3%
 THRUSTERS  . . . .  |*****                                              2.2%
 GEOMAGNETIC_FIELD   |*****                                              2.2%
 DEBUG_COLLECTOR  .  |****                                               2.1%
 MAGNETOMETER . . .  |****                                               1.7%
 SADA . . . . . .    |***                                                1.4%
 SOLAR_SYSTEM . . .  |***                                                1.2%
 MTA  . . . . . .    |**                                                 1.1%
 TTDM . . . . . .    |**                                                 1.0%
 TIMER  . . . . . .  |**                                                 0.9%
 DIRA . . . . . .    |*                                                  0.6%
 THRUSTER_COMMAND .  |*                                                  0.6%
 EARTH_ATMOSPHERE .  |*                                                  0.5%
 HARDWARE_UTILITIES  |*                                                  0.5%
 TACHOMETER . . . .  |*                                                  0.3%
 TOKEPL . . . . . .  |*                                                  0.3%
 IMAGER_DEVICE  . .  |*                                                  0.3%
 DSSA . . . . . .    |*                                                  0.3%
 SOUNDER_DEVICE . .  |*                                                  0.3%
 GYRO_CONVERTER . .  |*                                                  0.2%
                     +----------+----------+----------+----------+----------+
```

Figure A-2. Sample PCA Output

A.3 SECOND APPROACH—PIWG

Design alternatives not isolated to a particular part of the simulator were tested using the PIWG structure of measurements. The PIWG structure of measurements is based on the concept of a control loop and a test loop. The test loop contains everything in the control loop, plus one alternative to be measured. The CPU time is sampled before the execution of each loop and after many iterations of each loop. If the test loop time duration is not considered stable, the process is repeated with a greater number of iterations; this is accomplished through the presence of an outer loop surrounding the test and control loops. To be considered stable, the test loop time duration must be greater than a predefined minimum time. If this condition is met, the test loop time duration is compared against the control loop time duration, and the

number of iterations is compared against a predefined minimum number of iterations. If the test loop time is greater than the control loop time or the minimum number of iterations has been exceeded, the results are considered stable and the CPU time for the design alternative is calculated. The time for the alternative is the difference between the amount of CPU time taken for the control loop and the amount of CPU time taken for the test loop, divided by the total number of iterations performed. Collecting control loop and test loop CPU times, calculating design alternative times, and testing for stability were done using PIWG's Iteration package in the test drivers for this study.

All test drivers used in this study were called three times from a main driver routine so that the CPU time for a given design alternative could be averaged for more accuracy. All results were averaged and recorded using PIWG's I/O package and report generator procedure. A sample PIWG report is contained in Figure A-3.

```
Test Name:    Generic_A                    Class Name:  Matrix - Gen
CPU Time:       117.2  microseconds
Wall Time:      117.2  microseconds        Iteration Count:   128
                                           Number of samples: 3

Test Description:
Use of generic matrix processing
- Generic package for 3x3 matrix


Test Name:    Generic_C                    Class Name:  Matrix - Gen
CPU Time:       117.2  microseconds
Wall Time:      117.2  microseconds        Iteration Count:   128
                                           Number of samples: 3

Test Description:
Use of generic matrix processing
- NonGeneric package for 3x3 matrix
```

Figure A-3. Sample PIWG Report

As mentioned in Section A.1, some of the tests using this approach fall into the category of "synthetic benchmarks." The synthetic benchmarks in the PIWG test suite sample the CPU time before and after many iterations of the sequence of statements being tested and then divide the difference of these two times by the number of iterations. The synthetic benchmarks developed from this study use the batch job approach and obtain the CPU time from the log file generated when the benchmark is submitted to be run in batch.

GLOSSARY

ACM	Association for Computing Machinery
CPU	central processing unit
DEC	Digital Equipment Corporation
EMS	electronic mail system
ESA	Earth sensor assembly
EUVE	Extreme Ultraviolet Explorer
EUVEDSIM	EUVE dynamics simulator
EUVETELS	EUVE telemetry simulator
FDD	Flight Dynamics Division
GOADA	GOES attitude dynamics simulator in Ada
GOES	Geostationary Operational Environmental Satellite
GOESIM	GOES telemetry simulator
GRO	Gamma Ray Observatory
GRODY	GRO attitude dynamics simulator in Ada
GROSS	GRO attitude dynamics simulator in FORTRAN
GSFC	Goddard Space Flight Center
I/O	input/output
LRM	language reference manual
OOD	object-oriented development
OOP	object-oriented programming
PCA	performance coverage analyzer
PIWG	performance issues working group
SAMPEXTS	Solar, Anomalous, and Magnetospheric Particle Explorer telemetry simulator
SEL	Software Engineering Laboratory
SIGADA	special interest group on Ada
UARS	Upper Atmosphere Research Satellite
UARSTELS	UARS telemetry simulator
VAXset	VAX software engineering tools
VMS	virtual memory system

REFERENCES

1. *Ada Programming Language*, American National Standards Institute/Military Standard 1815A, January 1983 (ANSI/MIL-STD-1815A-1983)

2. Computer Sciences Corporation, SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby et al., December 1988

3. Goddard Space Flight Center, FDD/552-90/010, *Software Engineering Laboratory (SEL) Study of the System and Acceptance Test Phases of Four Telemetry Simulator Projects*, D. Cover, prepared by Computer Sciences Corporation, September 1990

4. —, FDD/552-90/045, *Extreme Ultraviolet Explorer (EUVE) Telemetry Simulator (EUVETELS) Software Development History*, E. Booth and R. Luczak, prepared by Computer Sciences Corporation, June 1990

5. Booch, G., *Object Oriented Design*, The Benjamin/Cummings Publishing Company, Inc.: Redwood City, California, ISBN 0-8053-0091, 1991

6. Goddard Space Flight Center, SEL-87-006, *Ada Style Guide*, E. Seidewitz et al., June 1986

7. Byrne, D., and R. Ham, *Results, Chapter 1, Ada Versus FORTRAN Performance Analysis Using the ACPS*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, New York, ISBN 0-89791-354, vol. 10, no. 3, Winter 1990

8. Goddard Space Flight Center, FDD/552-89/006, *Upper Atmosphere Research Satellite (UARS) Telemetry Simulator (UARSTELS) Software Development History*, E. Booth and R. Luczak, prepared by Computer Sciences Corporation, November 1989

9. Firesmith, D., "Two Impediments to the Proper Use of Ada," *ACM SIGAda Ada Letters*, Association for Computing Machinery, New York, New York, ISSN-076-721, vol. 7, no. 5, September/October 1987

10. Digital Equipment Corporation, *VAX Ada Run-Time Reference Manual (Version 2.0)*, May 1989

11. —, *VAX Ada Reference Manual (Version 2.0)*, May 1989

12. Burley, R., *Some Data from Ada Performance Study*, internal FDD memorandum, September 1990

13. Clapp, R., and T. Mudge, *Rationale, Chapter 1—Introduction*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, New York, ISBN 0-89791-354, vol. 10, no. 3, Winter 1990

14. Clapp, R., and T. Mudge, *Rationale, Chapter 3—The Time Problem*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, New York, ISBN 0-89791-354, vol. 10, no. 3, Winter 1990

15. Gaumer, D. and D. Roy, *Results, Reporting Test Results*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, New York, ISBN 0-89791-354, vol. 10, no. 3, Winter 1990

16. Digital Equipment Corporation, *Guide to VAX Performance and Coverage Analyzer*, June 1989

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-77-004, *A Demonstration of AXES for NAVPAK*, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, *GSFC NAVPAK Design Specifications Languages Study*, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-003, *Common Software Module Repository (CSMR) System Description and User's Guide*, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, *Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study*, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-008, *Cost and Reliability Estimation Models (CAREM) User's Guide*, J. F. Cook and E. Edwards, February 1981

SEL-81-009, *Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation*, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings From the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, *Software Engineering Laboratory (SEL) Compendium of Tools*, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-205, *Recommended Approach to Software Development*, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings From the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-906, *Annotated Bibliography of Software Engineering Laboratory Literature*, P. Groves and J. Valett, November 1990

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-006, *Monitoring Software Development Through Dynamic Variables*, C. W. Doerflinger, November 1983

SEL-83-007, *Proceedings From the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-101, *Manager's Handbook for Software Development, Revision 1*, L. Landis, F. McGarry, S. Waligora, et al., November 1990

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings From the Ninth Annual Software Engineering Workshop*, November 1984

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, *Proceedings From the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card, October 1986

SEL-86-006, *Proceedings From the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, *Data Collection Procedures for the Rehosted SEL Database*, G. Heller, October 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, S. DeLong, November 1987

SEL-87-010, *Proceedings From the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-003, *Software Management Environment (SME) Concepts and Architecture*, W. Decker and J. Valett, August 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/ Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-101, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1)*, M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Spiegel, February 1990

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler and K. Pumphrey, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. J. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-002, *Software Engineering Laboratory (SEL) Data and Information Policy*, F. McGarry, April 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. Booth and M. Stark, July 1991

SEL-RELATED LITERATURE

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics*, March 1981

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[2]Basili, V. R., R. W. Selby, Jr., and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., *Comparing the Effectiveness of Software Testing Strategies*, University of Maryland, Technical Report TR-1501, May 1985

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[5]Basili, V., and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D., and W. Agresti, "Resolving the Software Science Anomaly," *The Journal of Systems and Software*, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," *The Journal of Systems and Software*, June 1988

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

[5]Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

Hamilton, M., and S. Zeldin, *A Demonstration of AXES for NAVPAK*, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F., and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

National Aeronautics and Space Administration (NASA), *NASA Software Research Technology Workshop* (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[5]Ramsey, C., and V. R. Basili, *An Evaluation of Expert Systems for Software Engineering Management*, University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[8]Straub, P. A., and M. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software*, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, *NASA/SEL Data Compendium, Data and Analysis Center for Software*, Special Publication, April 1981

[5]Valett, J., and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experience With Syntax Editors," *Information and Software Technology*, April 1990

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.