AD-A250 183

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

(1)

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
1991

3. REPORT TYPE AND DATES COVERED
THESIS/XXXXXXXXXX

4. TITLE AND SUBTITLE
Execution Time Prediction of Ada Programs

5. FUNDING NUMBERS

6. AUTHOR(S)
Christopher A. Warack, Captain

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student Attending: University of Michigan

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA-91-134

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/CI
Wright-Patterson AFB OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW 190-1
Distributed Unlimited
ERNEST A. HAYGOOD, Captain, USAF
Executive Officer

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

14. SUBJECT TERMS

15. NUMBER OF PAGES
121

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

Execution Time Prediction of Ada Programs

Christopher A. Warack
Captain, US Air Force

1991

121 pages

Master of Science
University of Michigan

Specification of the timing properties of real-time systems is a fundamental part of their requirements. Analyzing the timing properties of the system's design and implementation is an important issue for the system developer. Timing analysis is necessary to determine the validity of a design or implementation in respect to the real-time specification.

Using timing schema and PERT networks, Ada program timing behavior can be analyzed. The use of PERT networks is simple but restricted to single processor systems. Replacing the PERT networks with a communicating real-time state machines model allows the analysis of Ada programs on multi-processor systems.

The technique is developed with examples and applied to a Macintosh IIsi programming environment. A foundation is laid for measuring how good a timing analysis prediction fits the implementation.

Bibliography

1. Shaw, A.C., *Reasoning About Time in Higher-Level Language Software.* IEEE Transactions on Software Engineering, 1989. 15(7): p. 875 - 889.

2. Park, C.Y. and A.C. Shaw. *A Source-Level Tool for Predicting Deterministic Execution Times of Programs*. Department of Computer Science, University of Washington, (Technical Report 89-09-12). September 13, 1989.

3. Shaw, A.C. *Towards a Timing Semantics For Programming Languages*. in *Third Annual Workshop, Foundations of Real-Time Computing*. 1990. Washington, DC: Office of Naval Research.

4. Goos, G., W.A. Wulf, A. Evans Jr., and K.J. Butler, ed. *DIANA: An Intermediate Language for Ada*. Lecture Notes in Computer Science, ed. G. Goos and J. Hartmanis. 1983, Springer-Verlag: Berlin. 201 pages.

5. Shaw, A.C. *Communiating Real-Time State Machines*. Department of Computer Science and Engineering, University of Washington, (Technical Report 91-08-09). August 1991.

6. Hoare, C.A.R., *An Axiomatic Basis for Computer Programming.* Communications of the ACM, 1969. **12** (10): p. 576-580.

7. Shaw, M. *A formal System for Specifying and Verifying Program Performance*. Department of Computer Science, Carnegie-Mellon University, (Technical Report CMU-CS-79-129). 21 June 1979.

8. Walden, E. and C.V. Ravishankar, *A Survey of Hard Real-Time Scheduling Algorithms.* unpublished draft, 1990..

9. Cornhill, D., *et al*. *Limitations of Ada for Real-Time Scheduling*. in *Proceedings of the International Workshop of Real-Time Ada Issues*. 1987. Moretonhampstead, Devon, UK: ACM SIGAda.

10. *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, Ada Joint Program Office, (January 1983).

11. Motorola, *MC68030 Enhanced 32-Bit Microprocessor User's Manual*. Third ed. 1990, Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

12. Park, C.Y. and A.C. Shaw. *Experiments With a Program Timing Tool Based on Source-Level Timing Schema*. in *IEEE Real-Time Systems Symposium*. 1990. Lake Buena Vista, Florida: IEEE Computer Society Press.

Execution Time Prediction of Ada Programs

by
Christopher Allen Warack

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer Science and Engineering)
in The University of Michigan
1991

Thesis Committee:

Professor Kang G. Shin, Chairman
Professor C.V. Ravishankar
Professor Stuart Sechrest

Abstract

Specification of the timing properties of real-time systems is a fundamental part of their requirements. Analyzing the timing properties of the system's design and implementation is an important issue for the system developer. Timing analysis is necessary to determine the validity of a design or implementation in respect to the real-time specification.

Using timing schema and PERT networks, Ada program timing behavior can be analyzed. The use of PERT networks is simple but restricted to single processor systems. Replacing the PERT networks with a communicating real-time state machines model allows the analysis of Ada programs on multi-processor systems.

The technique is developed with examples and applied to a Macintosh IIsi programming environment. A foundation is laid for measuring how good a timing analysis prediction fits the implementation.

Acknowledgement

In the short period of my graduate studies I was aided by several friends and associates. I cannot thank them all appropriately, but I would like to single out some of those whose help was critical to this work.

Foremost, I must thank my advisor, Professor Kang Shin. His help and patience has greatly broadened my understanding and appreciation of real-time systems. Secondly, I appreciate the efforts of the other members of my committee, Professors Stuart Sechrest and C.V. Ravishankar.

I would also like to thank Professor Alan Shaw whose work inspired this effort and who took time to answer my questions about his techniques. Major David Umphress at the Air Force Institute of Technology and William S. Salyers of Rational spent valuable portions of their time in getting me access to a Rational computer system. Mike Bonamassa of Rational lent me key material and advice on working with DIANA and tools on the Rational machine.

Finally, I cannot show enough appreciation for the patience, support, and love of my family. My wife, Karen, picked up the slack around the house after a full day of work so I could continue working into the evening. My two-year-old daughter, Kristina, made sure I did not get completely lost in my work by coming to the office door and asking Daddy for "hugs." They listened, helped, and kept a smile on my face.

Table of Contents

Chapter I

Introduction

Motivation

A gap often exists between stating the requirements for a real-time system and determining if a design matches those requirements. Timing requirements suffer much from lack of adequate attention during the design process since techniques to analyze and abstract timing characteristics are difficult to find or do not trace forward to the implementation. Since the timing characteristics are among the most critical in real-time systems, the ability to track these requirements throughout the development life-cycle is crucial.

The simplest development life-cycle model is the waterfall model. In this model, each activity — specification, design, implementation, and test — occurs sequentially after the completion of the preceding phase. In practice, the waterfall model creates artificial bottlenecks and places unrealistic constraints on the project. More practical developments, however, still conduct the same four basic activities [1], [2]. In them, however, the activities may occur in different order and repetitiously. Furthermore, different pieces of the project may exist in different activity phases.

The model for software maintenance is quite similar. The fielded system has completed all of the development stages. New parts and modified parts may exist in various stages of completion, however. These additions and modifications are certainly part of the system and cannot be developed completely independent from the fielded system. The key difference between software development and maintenance is that system design and implementation decisions may not be available to the maintainers. While doing the same type of thing, maintainers often have less information outside that embedded in the system.

For an analysis technique to be applicable to the entire project, then, it must be consistent across development activities. The technique must be compatible with analyzing the system's parts existing simultaneously in different stages of development. This requires the technique to apply during design as well as during and after implementation. The results must combine into project-wide results, and these results must relate to the requirements specified for the system.

Typically, real-time systems have finite worst-case response time and workload requirements for specified scenarios. Therefore, a system timing analysis tool must generate predicted response time and workload measures for the system through the various activities of the development life-cycle. Only with this type of support can system developers track and focus on timing requirements during system development and maintenance.

Objectives

This thesis describes a technique for analyzing execution times of Ada programs. The primary objective for development of the analysis technique is that it can apply consistently throughout the development life cycle and answer questions about response times and workload scenarios. Pyster points out that the "hardest parts of developing software are specifying and validating requirements and design" [3]. The primary objective of this thesis focuses on validating the timing properties of the design as well as the implementation. Secondary objectives include:

- Applicability of the technique to different development and target environments with parametric differences only,
- Limited restrictions on design methodology or implementation style, and
- Few, if any, restrictions on applicability to legitimate Ada programs.

The technique is based on source code timing schema [4-6]. The results of applying the schema is used to construct a dependency graph of events connected by code segments where the code segments are represented by their execution times. This graph can be manipulated to determine the worst-case path between two events. The resulting length of the path is the response time. Simultaneous solution for a scenario of events can determine the worst-case processor utilization necessary in a specified period of time, thus determining workload.

Organization

The thesis is organized as follows. Chapters II through IV develop the schema-based execution time analysis algorithm and concurrency extensions. Chapter V presents and summarizes experimental results.

Chapter II develops the background of execution time prediction and scheduling analysis. Existing prediction techniques are evaluated with the criteria described in the objective. Scheduling analysis relies on *a priori* knowledge of task time behavior. The tasks used in the scheduling literature are not necessarily the same as Ada tasks. Thus, a context is developed that connects traditional scheduling analysis and Ada timing analysis. This background sets the stage for development of the analysis technique.

The execution time prediction method develops in Chapter III. Timing schema are defined for a DIANA [7] representation of Ada programs. The schema is defined in terms of primitives that partition language primitive constructs (declarations, statements and expressions) into shared and branch-distinct portions. Furthermore, the schema defines where context switches may occur; these points are defined as events. Compiler analysis generates execution time bounds for each primitive. These steps provide the data for an algorithm to transform DIANA trees into the analysis graph. This graph has events for nodes and code sequences weighted by their execution time bounds for edges.

Chapter IV further develops the concurrency issues involved. It defines the Ada concurrency model. Some limitations on real-time programs naturally fall out of the application of the timing analysis technique to Ada. The next step is to manipulate the analysis graph to generate response time and workload values. Two techniques are described. One uses Communicating Real-Time State Machines (CRSM) [8]. The other uses PERT analysis techniques.

Experiments are designed to show that the calculated times do indeed bound execution times. Furthermore, simple experiments are hand analyzed for worst-case time and compared to the predicted time. Finally, the predicted, actual and hand-analyzed times are compared to provide a qualitative measure of the technique.

Chapter II

Execution Time Prediction and Scheduling

Introduction

Execution time prediction and scheduling analysis are relatively old problems. Yet, they are far from resolved problems. The standard techniques for determining execution time are primitive. This information, though, is critical to performing scheduling analysis. Only when the execution time of a given *task* is known, then scheduling analysis may be able to determine whether all deadlines can be satisfied. This chapter compares existing time prediction techniques to the criteria of life-cycle applicability, portability, and language limitations. It then discusses how the idea of a *task* in scheduling analysis relates to system design and implementation.

Traditionally, the execution time of a program is measured using instruction analysis of the underlying object code or through testing the actual execution time of the implementation. More advanced techniques are developed in [4, 9-14]. These are not used in practice, however, for one reason or another. By comparing these methods to some criteria defining the needs of real-time Ada program development, motivation develops for a technique derived from Shaw's timing schema [4].

In scheduling analysis *tasks* (italicized for distinction) are commonly defined as a tuple of an arrival time, a period or deadline, and a maximum execution time. This is an abstract notion that relates well to the context of many real-time systems — an event occurs, a *task* is generated to react to it and must complete prior to a deadline; or, an activity must occur periodically and complete prior to the end of the period. This notion of a *task* is very different from the Ada notion of a task. An Ada task is a program construct which exhibits concurrency. These distinct ideas of tasks are resolved by relating a *task* to Ada programming constructs.

Traditional Execution Time Prediction Methods

Two sections in Knuth [15] discuss "Analysis of an Algorithm" and "O-Notation." While asymptotic analysis is useful in making wide distinctions in efficiency, it does not relate directly to time. Deadlines are stated in microseconds or milliseconds, not in O-Notation. Two practical methods are widely used. Knuth also discusses the first of these, the hand analysis of machine object code. He combines this with asymptotic analysis to generate execution time predictions. In general, common sense and logic are used to derive meaningful information from the object code. The other widely-used method is benchmarking or test case monitoring.

The most obvious drawback of these techniques to life-cycle analysis of execution time is that they require object code exist before operating. Thus, they are of limited use during implementation and only fully useful after its completion. Hand analysis is also prone to be highly complex. Knuth's example [15, pp. 164 - 169] takes an 83 line assembly program and reduces it to a linear equation with six independent variables. To accomplish this required an application of Kirchoff's law to an earlier set of 15 variables and significant application knowledge. In large real-time programs, this complexity is overwhelming. On the other hand, testing suffers from lack of rigor. Unless careful analysis shows that the test cases generate a relationship to the worst case execution time for the code, then these results are not necessarily legitimate bounds. Possibly, the derived times may bound the execution time *most of the time*. This qualification, though, is not quantifiable and the consequences of failure may be too severe to rely on it. Thus, we learn from these techniques a need for analytic simplicity and logical rigor. Structured programming in higher-level languages was developed to simplify the logical complexity of unstructured and assembler code. Several researchers have turned toward analysis of the source code for timing information to benefit from its reduced complexity.

Mok's annotation technique [11] automates the hand analysis process described by Knuth. Like Knuth's technique it is limited to the implementation and post-implementation phases. The developer annotates the program. The annotated program is fed to a set of timing analysis tools and to a special compiler. The compiler is modified only in that it adds labels to various assembly instructions in the code generation phase. These labels do not affect the final object code that the assembler stage generates. Besides feeding the assembler stage, the annotated assembly code is also fed to the timing analysis tool. Using the labels, the annotated source code and assembly code are merged. The developer then works interactively with the timing tools to generate timing information about the program.

Many of the more advanced techniques, including Haase's guarded commands and PARCs [9], Halang's extensions to the PEARL programming language [10], Puschner and Koza's MARS-C language [12], and Kenney and Lin's FLEX [13], are based on theoretical languages or language extensions that are not used in practice. Thus, they are not portable as defined nor do they use Ada. Many of these extensions could be translated to Ada; however, Ada supersets are disallowed in Department of Defense projects and are discouraged in general. This raises the restriction that the technique not require language extensions or particular code generation behavior beyond that stated in the language definition. It is important to note that annotations like those used by Mok satisfy this restriction. Although they require support in the compiler, they do not affect the language definition.

Glicker and Hosch describe a system that uses symbolic execution to model the behavior of Ada programs [14]. It first determines the best and worst case threads of execution through the Ada task system given a set of preconditions. These threads are then measured directly using the target architecture. This method shows some promise although it can only apply to completed programs. Its success hinges on the adequacy of the symbolic execution stage. Tracing all threads through a program is naturally an $O(e^x)$ time activity. Good branch-and-bound heuristics must be applied to keep the process tractable.

Several of these techniques share common features. The most prevalent of these is analysis of the programming language constructs themselves to reason about the timing behavior of the program. This concept is distilled by Shaw in his timing schema approach. Shaw's work is done in C. Ada has several language constructs that make timing analysis easier and several more complex constructs than the simple statements and expressions allowed by C.

In summary, existing techniques are generally not applicable to large real-time Ada projects. Furthermore, they are not applicable throughout the development process. Mok's annotations could be extended to Ada's sequential language constructs. Its biggest short-coming is reliance on compiler support. It also raises some configuration management hurdles in ensuring that the data under analysis comes from the current code baseline. Jumping these hurdles is straight-forward with careful process management. Shaw's timing schema grant the developer less flexibility than Mok's for timing analysis, but do not rely on compiler support. Both techniques must be extended to handle Ada tasking constructs. This thesis' technique develops Shaw's timing schema, adding tasking support, while allowing some compiler-independent annotations or *assertions*.

Schema Based Timing Analysis

Timing schema are based on Hoare logic [16]. Hoare logic uses the notation {P}S{Q} to mean that given the conditions P immediately prior to execution of S results in condition Q upon completion of execution. An inherent assumption is that S does, in fact, complete. The change in time from executing statement S can be described with Hoare logic as follows $\{rt = x\}S\{rt = x + t(S)\}$, where rt represents the time, x represents the value of rt immediately prior to execution of S and t is a function which returns the execution time of S. The first application of Hoare logic like this is in a paper by Mary Shaw. [17]

The function t has as its domain the set of all possible programs and as its range the non-negative real numbers. A definition of t can consist of a set of axioms {(S,t)} relating all programs to their execution time. Since there are infinite programs, this approach cannot always work. However, t can be described with a finite set of rules for generating the infinite set of relations above. This set of rules is a schema.

The above is flawed in that t in the rule set must be a single value to define a well-formed function. For a single execution of a program, it will indeed take a specific time value to execute. In fact, the execution time of a given program may vary from one execution to another. There exist many reasons for this including differences in inputs, differences in machine state at the start of execution, inconsistencies in the machine clocking mechanism and differences in the machine-language instantiation of the program. Further discussion of these variabilities will occur in the section on compiler analysis in Chapter 3. Let t'(S,x) be the function which returns the single value for the execution time of S as indexed by x $\in$ {all possible execution conditions}. A more correct model of t, then, may be a random variable; thus t is a mapping from programs to random variables. It is adequate, however, to model t as a closed interval $T = [t_{min}, t_{max}]$ where $P[t'(S,x) \in T] = 1$ for all x. Therefore, timing analysis of a program S consists of computing t(S). This is done using a schema which generates a rule computing a closed time interval, T, for S.

Arithmetic and logical operations on values of T use a form of interval arithmetic. Additive operations are simply applied component-wise. Multiplicative operations are distributed, T **op** $x = [t_{min}$ **op** x, t_{max} **op** $x]$. Relational operations are applied to the t_{max} component first since the worst-case is the more important in real-time analysis. If the t_{max} components are equal, then comparison of the t_{min} components determines the result.

The schema are defined for the grammatical elements of a language. For instance, the schema for an if-statement, $\mathcal{T}$[`if-statement`], might be:

$\mathcal{T}$[`if-statement`] = [10,15] + $\mathcal{T}$[`boolean-exp`] + [min(([4,6] + $\mathcal{T}$[`then-part`]), ([2,4] + $\mathcal{T}$[`else-part`])), max(([4,6] + $\mathcal{T}$[`then-part`]), ([2,4] + $\mathcal{T}$[`else-part`]))]
`where if-statement ::= if boolean-exp then then-part else else-part;`

The schema rule's definition consists of constant parts like [10,15] and recursive invocation of other schema rules. These constant parts represent some basic computation to provide the language behavior for that construct. The logical basis of the above rule is that the system computes the boolean expression (boolean-exp) and spends time branching on the result ([10,15]). The branch with the bigger execution time is the worst case choice and the lesser execution time is the best case choice. The branch execution time consists of the time spent executing the statements in the branch (then-part or else-part) and time spent rejoining the main execution stream ([4,6] or [2,4]).

Primitive times like the constant parts described above are dependent on the underlying system consisting of the hardware, operating system, and compiler. The exact value of the primitives will differ from one underlying system to another, but it is always present in the schema since its existence derives from some computational need in the language definition. It may be possible for a particular system to compute a particular primitive in time $[0,\varepsilon]$, for a very small $\varepsilon > 0$, using special hardware or subsuming it in other primitives.

Eventually, any given program will reduce through the schema rules to a sum of primitive times. This is akin to parsing the language where non-terminals in the grammar are similar to schema rules and terminals are similar to primitive times. Computing the sum gives the resulting execution time of the code sequence. One inherent assumption throughout this discussion has been that the program has a single *thread* of execution. With multiple threads, timing schema analysis must be applied to each thread. The relationship between the threads must be addressed with other mechanisms. These mechanisms will be discussed in the next two chapters.

Scheduling Analysis

Scheduling is one of the main problems in the area of real-time systems. The problem is to determine if a given set of tasks can meet all of their deadlines on a given set of processors. Tasks in the sense of scheduling are described as a triple $\{a, c, d\}$ where a is the arrival time, c is the execution time required to complete the task, and d is the deadline for the task [18]. Knowing the execution time of a task, then, is critical to conducting scheduling analysis.

This notion of a task, however, is abstract. An Ada task, on the other hand, is a concrete programming construct. The two concepts do not necessarily relate directly. Consider an event-driven system. The response of the system to a certain input event is a task for scheduling purposes. Its arrival time is the time of the event. Its deadline is the response time specified for the system (derived from physical requirements or allocation of other timing requirements). The final component is the execution time. While the first two components are commonly defined in the system requirements or description, the execution time results from design and implementation. Timing analysis is the technique to determine the execution time.

During the requirements and early design phase, the software analyst should determine the events, deadlines, and event arrival scenarios the system will handle. Scheduling analysis is used to determine if the system can satisfy its constraints. Thus, execution times must be predicted as early in the design as possible. As design progresses and implementation begins, the system model is refined. Scheduling analysis continues to determine if the current model is viable. Thus, execution timing analysis must continue and hopefully improve through the process. During validation, scheduling analysis with inputs from timing analysis is used to determine whether or not the system as implemented can satisfy its constraints in all cases. Testing is not satisfactory in many cases since it may not test the worst case conditions that may be encountered.

Scheduling analysis, however, depends on the scheduling strategy implemented by the system. A rate monotonic system can be implemented using Ada tasking and priorities. Ada, however, is currently prone to priority inversion [19]. Many scheduling approaches are available in any programming language by implementing a scheduler as part of the system. Cyclic executives are also popular with real-time developers [20].

In summary, it is important to keep distinct the concept of scheduling *tasks* and the Ada task constructs. While it is possible, Ada tasks do not map well to current scheduling strategies. Scheduling strategies can be built into a system, however, and Ada tasks used within the implementation of that system separate from scheduling policies.

Discussion

Timing Analysis using Shaw's schema techniques and Ada satisfies the stated objectives. When used with scheduling analysis, timing analysis can be used to show that a system satisfies its real-time requirements. Regardless, timing analysis can characterize program timing behavior to give system developers information for design, implementation and verification decisions.

Several existing timing analysis techniques only apply to completely implemented systems. The timing schema approach outlined above can be used with an Ada program design language (PDL) to provide timing analysis throughout the development and with varying stages of system completion. This lets the developer identify and track or correct problems early when they are cheapest to fix.

Chapter III

Timing Schema and Events

Introduction

The first step in timing analysis is transforming the Ada programs into timing graphs. The transformation is based on a DIANA representation of these programs. A schema rule is defined for each type of DIANA node. The resulting timing graph may be as simple as a single edge, representing a simple sequential program. It may also be very complex containing several nodes and branching alternatives. The generation of these graphs are discussed. The analyst has some control over the graphs through the use of assertions.

DIANA Representation of Ada Programs

DIANA is an abstract data type for representing Ada programs [7]. A DIANA object is mathematically modelled as an attributed tree. The tree represents a normalized form of a corresponding Ada program. It also guarantees that a given Ada object has only one defining occurrence; and the defining occurrence is an attribute of the other occurrences. Using DIANA takes care of the complicated Ada parsing and static semantic analysis. The tree model is easily manipulated.

A given node of the DIANA tree is defined in terms of the attributes it has. A node has a structural arity in the set {0,1,2,3,n} and has zero or more lexical, semantic, and code attributes. Additional attributes may exist as needed by an application; however, these are not standard and cannot be relied on. The structural arity denotes the branching of each node to other nodes. The other attributes may provide numeric or textual information or be semantically related nodes. By sharing identical nodes, the tree becomes a directed acyclic graph (DAG). The DAG model is identical to the tree model except that it allows replication to be eliminated.

For example, the simple program below simulates rolling x n-sided dice where x and n are supplied by the caller. This converts to the DIANA DAG described following it. The format for a DAG node is `name`[1] `: node_type [ attributes ]`. Attributes are juxtaposed pairs of the form `attribute_name attribute_value`. Multiple attributes are separated by semicolons.

```
function ROLL_DICE (NUM_SIDES, NUM_DICE : in INTEGER)
  return INTEGER is
   subtype DIE_RANGE is INTEGER 1 .. NUM_SIDES;
   A_DIE : DIE_RANGE;
   TOTAL : INTEGER    := 0;
begin
   for A_ROLL in 1 .. NUM_DICE loop
      A_DIE := INTEGER(RANDOM * NUM_SIDES) + 1;
      TOTAL := TOTAL + A_DIE;
   end loop;
   return TOTAL;
end ROLL_DICE;
```

Figure 1: Sample Ada Program

1 Nodes with names like PDx are part of the Ada package "Standard" provided as part of the compiler environment. These names are used consistently with the example in [7].

```
A1   :  comp_unit             [ as_pragma_s ^A2;
                                as_context ^A3;
                                as_unit_body ^A4 ]
A2   :  pragma_s              [ as_list < > ]
A3   :  context               [ as_list < > ]
A4   :  subprogram_body       [ as_designator ^A5;
                                as_header ^A6;
                                as_block_stub ^A7 ]
A5   :  function_id           [ lx_symrep "ROLL_DICE";
                                sm_spec ^A6
                                sm_body ^A7
                                sm_location void ]
A6   :  function              [ as_param_s ^A8
                                as_name ^A13 ]
A7   :  block                 [ as_item_s ^A14;
                                as_stm_s ^A29;
                                as_alternative_s ^A54 ]
A8   :  param_s               [ as_list < ^A9 > ]
A9   :  in                    [ as_id_s ^A10;
                                as_name ^A13;
                                as_exp_void void ]
A10  :  id_s                  [ as_list < ^A11 ^A12 > ]
A11  :  in_id                 [ lx_symrep "NUM_SIDES";
                                sm_init_exp void;
                                sm_obj_type ^PD9 ]
A12  :  in_id                 [ lx_symrep "NUM_DICE";
                                sm_init_exp void;
                                sm_obj_type ^PD9 ]
A13  :  used_name_id[2]       [ lx_symrep "INTEGER";
                                sm_defn ^PD8 ]
A14  :  item_s                [ as_list < ^A15 ^A16 ^A17 > ]
A15  :  subtype               [ as_id ^A18;
                                as_constrained ^A19 ]
A16  :  var                   [ as_id_s ^A23;
                                as_type_spec ^A25;
                                as_object_def void ]
A17  :  var                   [ as_id_s ^A26;
                                as_type_spec ^A13;
                                as_object_def ^A28 ]
A18  :  subtype_id            [ lx_symrep "DIE_RANGE";
                                sm_type_spec ^A19 ]
A19  :  constrained           [ as_name ^A13;
                                as_constraint ^A20;
                                sm_type_struct ^A19;
                                sm_base_type ^PD9;
                                sm_constraint ^A20 ]
A20  :  range                 [ as_exp1 ^A21;
                                as_exp2 ^A22 ]
A21  :  numeric_literal       [ lx_numrep "1";
                                sm_exp_type ^PD9;
                                sm_value 1 ]
A22  :  used_object_id        [ lx_symrep "NUM_SIDES";
                                sm_exp_type ^PD9;
                                sm_defn ^A11 ]
A23  :  id_s                  [ as_list < ^A24 > ]
```

2 Note that this node is heavily reused in the structural DAG. This is not surprising since it represents the type integer.

```
A24 : var_id              [ lx_symrep "A_DIE";
                            sm_obj_type ^A19;
                            sm_address void;
                            sm_obj_def void ]
A25 : used_name_id        [ lx_symrep "DIE_RANGE";
                            sm_defn ^A18 ]
A26 : id_s                [ as_list < ^A27 > ]
A27 : var_id              [ lx_symrep "TOTAL";
                            sm_obj_type ^PD9;
                            sm_address void;
                            sm_obj_def ^A28 ]
A28 : numeric_literal     [ lx_numrep "0";
                            sm_exp_type ^PD9;
                            sm_value 0 ]
A29 : stm_s               [ as_list < ^A30 ^A53 > ]
A30 : loop                [ as_iteration ^A31;
                            as_stm_s ^A36 ]
A31 : for                 [ as_id ^A32;
                            as_dscrt_range ^A33 ]
A32 : iteration_id        [ lx_symrep "A_ROLL";
                            sm_obj_type ^PD9 ]
A33 : constrained         [ as_name ^A13;
                            as_constraint ^A34;
                            sm_type_struct ^A33;
                            sm_base_type ^PD9;
                            sm_constraint ^A34 ]
A34 : range               [ as_exp1 ^A21;
                            as_exp2 ^A35 ]
A35 : used_object_id      [ lx_symrep "NUM_DICE";
                            sm_exp_type ^PD9;
                            sm_defn ^A12 ]
A36 : stm_s               [ as_list < ^A37 ^A49 > ]
A37 : assign              [ as_name ^A38;
                            as_exp ^A39 ]
A38 : used_object_id      [ lx_symrep "A_DIE";
                            sm_exp_type ^A19;
                            sm_defn ^A24 ]
A39 : function_call       [ as_name ^A40;
                            as_param_assoc_s ^A41 ]
A40 : used_bltn_op        [ lx_symrep "+";
                            sm_operator BINARY_PLUS ]
A41 : param_assoc_s       [ as_list < ^A42 ^A21 > ]
A42 : conversion          [ as_name ^A13;
                            as_exp ^A43 ]
A43 : function_call       [ as_name ^A44;
                            as_param_assoc_s ^A45 ]
A44 : used_bltn_op        [ lx_symrep "*";
                            sm_operator MULTIPLY ]
A45 : param_assoc_s       [ as_list < ^A46 ^A22 > ]
A46 : function_call       [ as_name ^A47;
                            as_param_assoc_s ^A48 ]
A47 : used_object_id      [ lx_symrep "RANDOM";
                            sm_exp_type ...;
                            sm_defn ... ]3
A48 : param_assoc_s       [ as_list < > ]
```

3 This is an external function with nodes outside the immediate program. These are elided for conciseness.

```
A49  :  assign               [ as_name ^A50;
                               as_exp ^A51 ]
A50  :  used_object_id       [ lx_symrep "TOTAL";
                               sm_exp_type ^PD9;
                               sm_defn ^A27 ]
A51  :  function_call        [ as_name ^A40;
                               as_param_assoc_s ^A52 ]
A52  :  param_assoc_s        [ as_list < ^A50 ^A38 > ]
A53  :  return               [ as_exp_void ^A50 ]
A54  :  alternative_s        [ as_list < > ]
```

Figure 2: DIANA Representation of the sample program

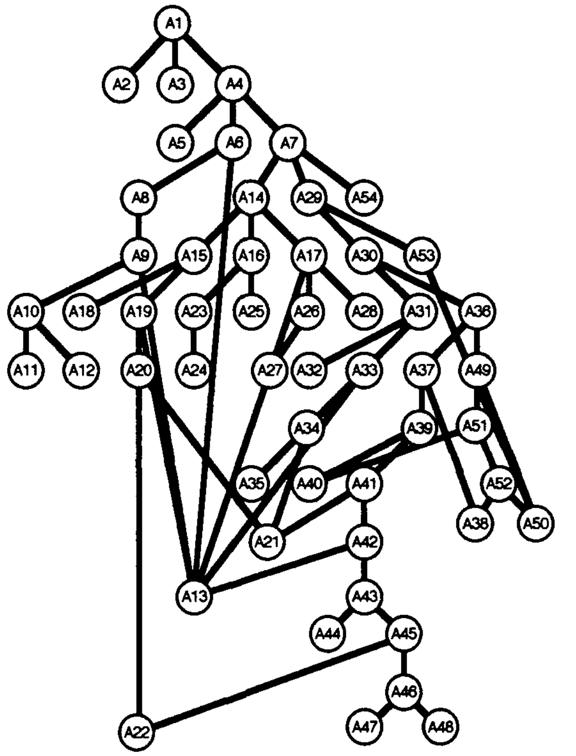


Figure 3: DAG of structural DIANA nodes for Sample Program

Timing Analysis Transformation Algorithm

The basic algorithm is to transform the DIANA object representing an Ada program into a graph. The edges of the graph represent sequential portions of the Ada program. The vertices represent potential context switches. The graph is built by traversing the tree depth-first or "bottom-up." As each node is traversed a subgraph is created based on the type of node and the subgraphs of its children. A simple implementation is to apply the schema function to the root node and using the recursive nature of the schema to traverse the tree. The traversal of a node is dependent on its type and structural attributes only. Its schema, however, may use semantic information in computing bounds and values.

The timing graph created by this process consist of edges weighted with execution times and vertices to connect edges. Multiple edges leaving a single vertex represent branching dependent on task synchronization. When constructed into a system network, only one branch in an instance will be utilized; the others are discarded.

A delay edge denotes a constraint that a certain time must pass between two vertices before execution can continue. A context switching node is so marked where a context switch may occur. This is done solely for the purposes of calculating the potential number of context switches in the resulting system model.

Timing Schema for DIANA Objects

The 170 different DIANA node types are listed below. Each entry includes the structural and key semantic attributes as well as the schema for computing the worst and best case time bounds for that node type. Several auxiliary functions are used to simplify the schema. These include:

- Store(Node): Determines the proper primitive time to store a value in an object by examining the type of the object.
- Access(Node): Determines the proper primitive time to access a value in an object by examining the type of the object.

- Init(Node): Determines the proper primitive time to initialize a value in an object by examining the type of the object.

- Save: This function updates the library of computed timing graphs with the graph computed for some subprogram, task, package, or generic declaration. These graphs are used by the Insert function or by the analyst in constructing a system network as described in the next chapter. Save "returns" the value [0,0]; that is, in storing the graph designated, this graph is not included in the computed time of the declaration block in which it occurs. This follows since it is not executed at time of declaration. Some elaboration time may be included in the schema in addition to the save, however. This does, indeed, execute during elaboration of the execution block.

- Stop: Halt computation within an enclosing stm_s type node. This occurs when an unconditional change in control flow is encountered in a sequence (i.e., return, goto, or unconditional exit).

- Abort: Ignore this path (stm_s node) since it contains a raise or abort statement. Currently raise and abort are restricted to use with error conditions. Error conditions are not analyzed by the technique at this time.

- Insert(Name): Insert the graph for a subprogram at this point in the current graph. Look for a recursion assertion if it is the same subprogram.

- Node(Name): End the current edge. Create a vertex. Start a new edge(s). With rendezvouses, the node will be linked to a corresponding node in another task, so one edge will be missing.

- Delay(Duration): Create a pair of nodes with a delay edge between them of the duration given.

- Activate(Task[s]): Create a context-switching node at this point preceded by the timing primitive P(activation) and attach Task[s] (as well as continuing the current thread).

- Queue-Activate(Task[s]): Like Activate except that the node is inserted after completing all processing of the current declarative block.

- ConstraintCheck(Node): Determine the proper primitive time for a constraint check on the type of the given node.

- Print(String): Print the given string on the analyzer's error output stream.

- Range(DSCRT_RANGE): Determines the lowest and highest possible values of the range.

Node Type	Arity	Schema
abort	1	Abort -- abort statements are treated as errors at this point and are not candidates for a min or max path.
accept	3	P(accept) + Node("start "&as_name) ; Node("Begin"&as_name) + P(rendezvous) + T(as_stm_s) + P(accept_end) + Node("end "&as_name)
access	1	T(as_constrained)
address	2	[0,0] -- representation clause affects compilation only
aggregate	n	$\left(\sum_{i=1}^{n} T(as\_list[i])\right)$ + Store(sm_exp_type)
alignment	2	[0,0] -- representation clause affects compilation only
all	1	[0,0];
allocator	1	T(as_exp_constrained) + if sm_exp_type = task_spec then Activate(task) else Store(sm_exp_type) + P(allocate_mem(Size(sm_exp_type)) Note: Records and arrays with task components need a hybrid of the if-statement above to apply.
alternative	2	-- subsumed in schema for case node -- ignored as attribute of exception part of blocks
alternative_s	n	-- subsumed in schema for case node -- ignored as attribute of exception part of blocks
and_then	0	P(and_then)
argument_id	0	[0,0] -- identifier "symbol table" entry
array	2	T(as_dscrt_range_s) + T(as_constrained)
assign	2	T(as_name) + T(as_exp) + Store(as_name) + ConstraintCheck(as_name) + if as_exp = used_object_id then Access(as_exp.sm_exp_type) else [0,0]
assoc	2	T(as_actual)
attr_id	0	[0,0] -- identifier "symbol table" entry
attribute	2	T(as_name) + P("attr_" & as_id.sm_defn.lx_symrep) -- attribute execution times are pre-defined
attribute_call	2	T(as_exp) + T(as_name) -- as_name is a node of type attribute
binary	3	[T(as_exp1), T(as_exp1) + T(as_exp2)] + T(as_binary_op)
block	3	T(as_item_s) + T(as_stm_s) -- as_alternative_s represents exception handlers
box	0	[0,0] -- generic subprogram formal option
case	2	T(as_exp) + P(case) + [min(choices),max(choices)] choices = {T(as_alternative_s.as_list[n].as_stm_s) : $1 \le n \le \lvert as\_list \rvert$} -- as_alternative_s.as_list[n].as_choice_s must be static and is ignored
choice_s	n	$\sum_{i=1}^{n} T(as\_list[i])$
code	2	T(as_exp) -- machine dependent code insertion -- execution time bounds must be specified with a time assertion
comp_id	0	[0,0] -- identifier "symbol table" entry
comp_rep	3	[0,0] -- representation clause affects compilation only
comp_rep_s	n	[0,0] -- representation clause affects compilation only
comp_unit	3	T(as_unit_body) -- as_context and as_pragma_s set up environment only
compilation	n	(map T as_list) = T(hd(as_list));(map T tl(as_list)) -- list of all compilation units in the system
cond_clause	2	-- subsumed in schema for "if" node
cond_entry	2	P(cond_entry) + Node1 + Delay(e) + T(as_stm_s2) + Node2; -- Delay(e) prejudices choice Node1 + T(as_stm_s1) + Node2; -- first stm is entry_call
const_id	0	[0,0] -- identifier "symbol table" entry
constant	3	T(as_type_spec) + T(as_object_def)
constrained	2	T(as_constraint)
context	n	[0,0] -- set's up environment, no cost

conversion	2	case as_name.sm_defn.sm_type_spec, as_exp.sm_type_spec real to int: P(convert_float2int) or P(convert_fixed2int) int to real: P(convert_int2float or P(convert_int2fixed) real to real: P(convert_fixed2float) or P(convert_float2fixed) derived : P(convert_derived) array: P(convert_array) others: [0,0] -- not changing base type + if subtype then Constraint_Check(sm_exp_type) else [0,0] + T(as_exp)
decl_s	n	$\sum_{i=1}^{n}$ T(as_list[i]) + Init(as_list[i])
def_char	0	[0,0]; -- never called since enum_literal_s is always [0,0] Access = P(num_access)
def_op	0	[0,0] -- operator “symbol table” entry
deferred_constant	2	[0,0] -- any cost will be incurred with full declaration
delay	1	T(as_exp) + P(delay) + Delay((sm_value or max(subtype range)) + P(delay_cap))
derived	1	T(as_constrained)
dscrmt_aggregate	n	$\sum_{i=1}^{n}$ T(as_list[i])
dscrmt_id	0	[0,0] -- identifier “symbol table” entry
dscrmt_var	3	T(as_object_def)
dscrmt_var_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
dscrt_range_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
entry	2	T(as_dscrt_range_void) + T(as_param_s)
entry_call	2	T(as_name) + T(sm_normalized_param_s) + P(queue_entry) + Node(“start “&as_name) ; Node(“end “&as_name) + [0,0]
entry_id	0	[0,0] -- identifier “symbol table” entry
enum_id	0	[0,0]; -- never called since enum_literal_s is always [0,0]
enum_literal_s	n	[0,0] -- list of def_char and enum_id; each static so [0,0] elaboration time
exception	2	[0,0] -- declares exception names
exception_id	0	[0,0] -- identifier “symbol table” entry
exit	2	if as_exp_void = void then Stop* else T(as_exp_void) -- computes condition, but doesn't affect number of loop iterations -- *a branch containing unconditional exit can only be executed once, -- so normally cannot be worst case branch[4]
exp_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
fixed	2	T(as_range_void) -- as_exp is static
float	2	T(as_range_void) -- as_exp is static
for	2	T(as_dscrt_range)
formal_dscrt	0	[0,0] -- place holder for generic type parameters
formal_fixed	0	[0,0] -- place holder for generic type parameters
formal_float	0	[0,0] -- place holder for generic type parameters
formal_integer	0	[0,0] -- place holder for generic type parameters
function	2	T(as_param_s)

[4] Can be best case, however. Furthermore, unusual conditions such as an exit branch which is much more computationally intensive than other branches through the loop or a loop which only executes a small number of times may be a worst case. A more thorough analysis may determine this.

function_call	2	if as_name is used_bltn_id or used_bltn_op then
		check sm_value for static expression and treat as folded constant
		if one exists, otherwise P(*bltn*) + T(sm_normalized_param_s)
		else
		P(function_call) + T(sm_normalized_param_s) + Insert(as_name)
		+ P(function_end)
		-- sm_normalized_param_s includes default params

function_id	0	[0,0] -- identifier "symbol table" entry
generic	3	save id(as_id) = T(as_generic_header) -- as_generic_param_s is used to define quasi-primitives to be replaced with actual times when instantiated.
generic_assoc_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
generic_id	0	[0,0] -- identifier "symbol table" entry
generic_param_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
goto	1	Print("Warning: Unstructured statement <goto> cannot be analyzed") + Stop -- if a forward goto, this will compute correctly; if it causes a loop, then the analysis is bad.
id_s	n	[0,0];
if	1	$\text{choices} = \left\{ \left(\sum_{i=1}^{n} \text{T(as\_list[i].as\_exp\_void)} + \text{P(else)} \right) + \text{T(as\_list[n].as\_stm\_s)} \; : \; 1 \le n \le \lvert \text{as\_list} \rvert \right\}$ P(if) + [min(choices),max(choices)]
in	3	if as_exp_void ≠ void then T(as_exp_void) + P(default_param)
in_id	0	[0,0] -- identifier "symbol table" entry
in_op	0	P(in)
in_out	3	[0,0] -- cannot have default parameters
in_out_id	0	[0,0] -- identifier "symbol table" entry
index	1	[0,0] -- as_name is an uconstrained type.
indexed	2	T(as_name) + T(as_exp_s) ;
inner_record	n	$\sum_{i=1}^{n}$ T(as_list[i])
instantiation	2	T(as_generic_assoc_s)
integer	1	T(as_range);
item_s	n	$\sum_{i=1}^{n}$ T(as_list(i))
iteration_id	0	[0,0] -- identifier "symbol table" entry
label_id	0	[0,0] -- identifier "symbol table" entry
labeled	2	T(as_stm)
loop	2	if as_iteration = void or as_iteration = while then T(as_iteration) + P(loop) + LOOP_ASSERTION * (T(as_stm_s) + T(as_iteration) + P(iter)) - P(iter) + P(loop_end) -- if no loop assertion then if no nodes in as_stm_s -- then print "Unbounded Loop" + Stop -- else unroll as far as necessary else T(as_iteration) + P(for_loop) + Range(as_iteration.as_dscrt_range) * (T(as_stm_s) + P(for_iter)) - P(for_iter) + P(for_end)
l_private	0	[0,0]
l_private_type_id	0	[0,0] -- identifier "symbol table" entry
membership	3	T(as_exp) + T(as_type_range) + T(as_membership_op)

name_s	n	$\sum_{i=1}^{n}$ T(as_list(i))
named	2	T(as_choice_s) + T(as_exp)
named_stm	2	T(as_stm)
named_stm_id	0	[0,0] -- identifier "symbol table" entry
no_default	0	[0,0] -- generic subprogram formal option
not_in	0	P(not_in)
null_access	0	[0,0]
null_comp	0	[0,0]
null_stm	0	P(null)
number	2	[0,0] -- static numeric constant
number_id	0	[0,0] -- identifier "symbol table" entry
numeric_literal	0	if sm_value ≤ SMALL_VAL then P(small_*num*_literal_access) else P(*num*_literal_access) -- where *num* is int, float or fixed
or_else	0	P(or_else)
others	0	[0,0]
out	3	[0,0] -- cannot have default parameters
out_id	0	[0,0] -- identifier "symbol table" entry
package_body	2	save id(as_id & "'body") = P(package_elaboration) + T(as_block_stub)
package_decl	2	save id(as_id) = T(as_package_def) + P(package_elaboration) -- if as_package_def is rename or instantiation then use other values
package_id	0	[0,0] -- identifier "symbol table" entry
package_spec	2	T(as_decl_s1) + T(as_decl_s2)
param_assoc_s	n	$\sum_{i=1}^{n}$ T(as_list[i]) -- actual parameter lists
param_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
parenthesized	1	T(as_exp)
pragma	2	[0,0] -- compiler directive may change global params, but no code gen Timing Assertion pragmas are handled, of course.
pragma_id	0	[0,0] -- identifier "symbol table" entry
pragma_s	n	[0,0] -- compiler directive may change global params, but no code gen
private	0	[0,0]
private_type_id	0	[0,0] -- identifier "symbol table" entry
proc_id	0	[0,0] -- identifier "symbol table" entry
procedure	1	T(as_param_s)
procedure_call	2	T(sm_normalized_param_s) + P(procedure_call) + Insert(as_name) + P(procedure_end) -- sm_normalized_param_s includes default params
qualified	2	T(as_exp) -- as_name only used by compiler
raise	1	Abort -- exceptions are treated as errors at this point and are not candidates for a min or max path.
range	2	T(as_exp1) + T(as_exp2)
record	n	$\sum_{i=1}^{n}$ T(as_list[i])
record_rep	3	[0,0] -- representation clause affects compilation only
rename	1	[0,0]
return	1	T(as_exp_void) + Store(function_call) + Stop
reverse	2	T(as_dscrt_range)
select	2	P(select) + $\sum_{i=1}^{\lvert as\_select\_clauses\_s \rvert}$ T(as_select_clauses_s.as_list[i].as_exp_void) -- comp guards + {map (λx.Node1 + T(x.as_stm_s) + Node2 , as_select_clauses_s.as_list) and if ∣as_stm_s∣ > 0 then Node1 + T(as_stm_s) + Node2}
select_clause	2	-- subsumed in schema for select node

select_clause_s	n	-- subsumed in schema for select node

selected	2	if as_name.sm_obj_type is a record then
		if it has a variant then
		T(as_name) + P(variant_tag_check) + T(as_designator_char)
		else
		T(as_name) + T(as_designator_char);
		else -- it is an expanded name
		T(as_name) + T(as_designator_char);

simple_rep	2	[0,0] -- representation clause affects compilation only
slice	2	T(as_name) + T(as_dscrt_range); size = | as_dscrt_range | * sm_exp_type.as_constrained.cd_impl_size
stm_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
string_literal	0	[0,0];
stub	0	[0,0] -- Used for separate compilation purposes only
subprogram_body	3	save id(as_designator) = T(as_block_stub) + T(as_header)
subprogram_decl	3	if as_subprogram_def ≠ void then save id(as_designator) = T(as_subprogram_def) + T(as_header)
subtype	2	T(as_constrained)
subtype_id	0	[0,0] -- identifier "symbol table" entry
subunit	2	save id(as_name) = T(as_subunit_body)
task_body	2	P(task_body_elab) + {save id(as_id & "'body") = T(as_block_stub)}
task_body_id	0	[0,0] -- identifier "symbol table" entry
task_decl	2	P(task_spec_elab) + {save id(as_id) = T(as_task_def)} + Queue_Activate(as_id)
task_spec	1	T(as_decl_s) -- Activated by allocators or declarations (if declaration, queue activate it)
terminate	0	P(terminate)
timed_entry	2	P(timed_entry) + Node1 + T(as_stm_s2) + Node2; -- first stm is a delay stm Node1 + T(as_stm_s1) + Node2; -- first stm is entry_call
type	3	T(as_type_spec) + T(as_dscrmt_var_s) -- if task type need to save it as well
type_id	0	[0,0] -- identifier "symbol table" entry
universal_fixed	0	[0,0];
universal_integer	0	[0,0];
universal_real	0	[0,0];
use	n	[0,0] -- controls visibility of ids; no code generated
used_bltn_id	0	[0,0]
used_bltn_op	0	[0,0]
used_char	0	[0,0];
used_name_id	0	[0,0]
used_object_id	0	[0,0];
used_op	0	[0,0]
var	3	| id_s | * [T(as_type_spec) + T(as_object_def) + if as_object_def = void then init(as_name) else store(as_name) + if as_type_spec = task_spec then Queue_Activate(as_id)] -- similarly for components of as_type_spec...
var_id	0	[0,0] -- identifier "symbol table" entry
variant	2	T(as_choice_s) + T(as_record)
variant_part	2	T(as_variant_s)
variant_s	n	$\sum_{i=1}^{n}$ T(as_list[i])
void	0	[0,0] -- void attribute; no code or semantic value
while	1	T(as_exp)
with	n	[0,0] -- controls visibility of ids; no code generated

Figure 4: Timing Schema for DIANA Nodes by Node Type

Notes:

- Schema with nodes are also described graphically in the next section.

- Expressions may be static. In this case they are evaluated during compilation and a value for the attribute sm_value is added in the DIANA representation. In this case the expression is handled as a constant object of the appropriate type. In other words, all nodes which can be expressions are first checked for the existence of an sm_value attribute before proceeding with application of the normal schema. These nodes include conversion, qualified, parenthesized, aggregate, binary, membership, indexed, slice, selected, all, attribute, attribute_call, and function_call.

- In loops the phrase n * T(x) is equivalent to unrolling the loop. This is only significant in cases where T(x) introduces nodes. If T(x) is simply additional edge weight, then n*T(x) can be directly computed using multiplication.

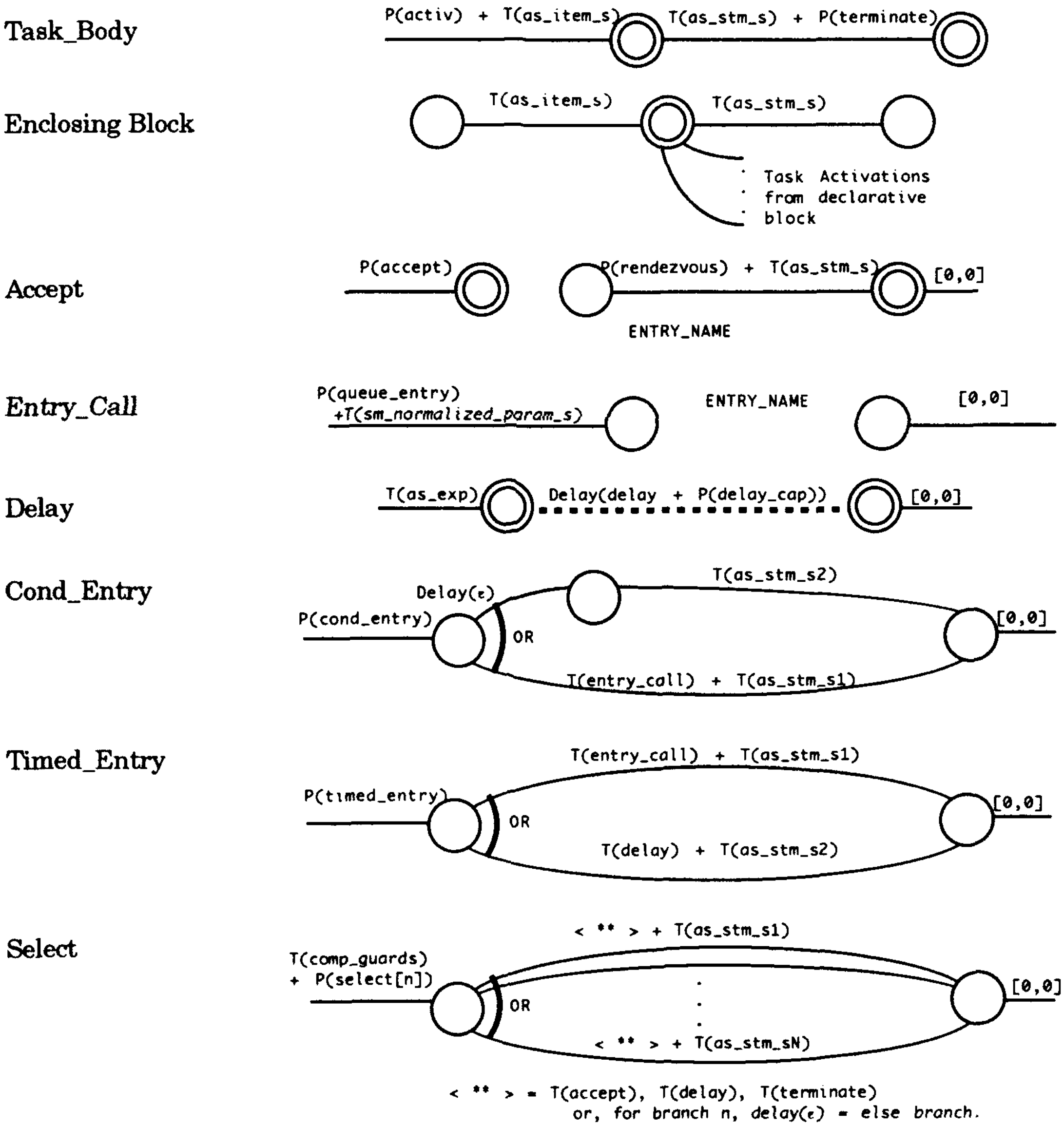
- Similarly, branching statements like if and case must be graphically represented if they contain nodes. This is illustrated in the next section.

- The abort statement non-cooperatively cuts off a task from further rendezvous and "marks" it for termination. This is normally used to recover from an error state. In any case, analysis stops on encountering an abort statement (like it does on raise statements) and chooses another parallel path as the worst or best case.

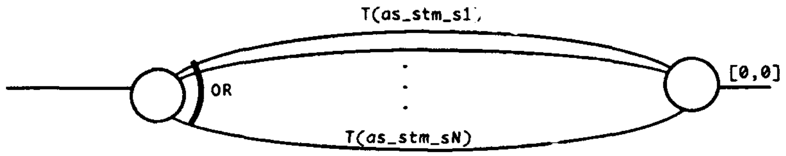
- Some schema of the form [0,0] are actually unreachable in computing the schema formula as defined. In general these are "ID nodes" which represent an identifier or operator of some sort. These nodes are very important in the computation of the auxiliary functions like store, access and init. ID nodes contain the semantic type information these auxiliary functions need. ID nodes are always leaves (i.e., they are never internal nodes) and are meaningful only within the context they appear. Therefore, the schema of their parent node normally include any primitives that context may induce as well as generating any auxiliary function computations necessary.

Event Structure for DIANA Objects

The potential context switches, or events, are introduced by certain Ada program constructs. The corresponding DIANA node types are listed below with a graphic description of the transformation involved. Figure 7 in the next chapter illustrates the transformation with some examples.  represents a vertex where a context switch may occur. Other vertices are used to connect edges and to gather alternative choices. Loops containing the following structures are unrolled completely if bounded and as far as necessary if unbounded.



If or Case containing one of the above



branch 1 through n-1 have some sort of node structure, branch n is computed from all of the *sequential* branches as is normally done. If no branch is purely sequential, all n branches have nodes.

In combining the timing graphs into a system network, entry_calls and accepts are *stitched* together like shown in the diagram below. Constructs which have alternative edges are instantiated with exactly one of those edges. Where the choice matters, careful selection must be made based on the analysis being performed. For instance, the worst case for a cond_entry in an unbounded loop is to always choose the *else* part. This results in an infinite chain of *else*'s. (This also illustrates why it is generally bad programming practice to use a conditional entry in an unbounded loop). Most of the time, however, the choices are evident.

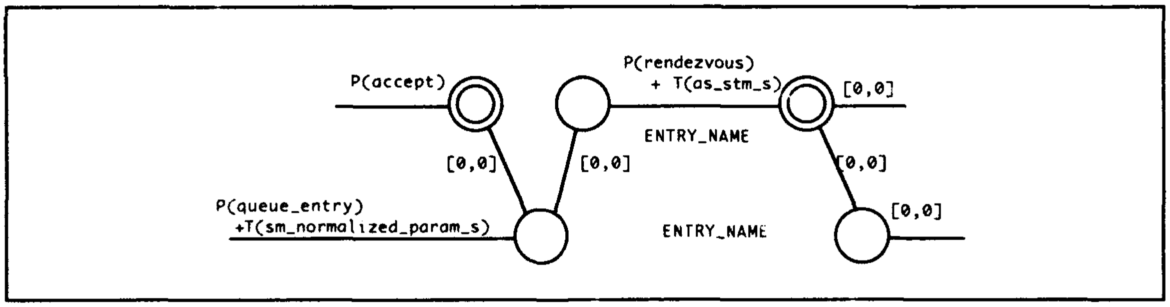


Figure 5: Stitching Together an Accept Statement and Entry Call

Assertions

An assertion is simply a statement. Classically, a software developer uses assertions to make claims about the state or nature of the program at a particular point in the code. Often these assertions are embedded as comments in the code itself. A small number of programming languages like Eiffel include certain assertions as part of their syntax [21]. In some cases, an automated tool may process these assertions to generate more powerful claims. This basic idea is applicable to timing analysis. In fact, it grants much of the power for analyzing designs or incomplete code segments. Assertions may be used to bound unbounded loops or recursion, to specify the length of time some code will take without specifying the code itself, and to mark relevant points in the analysis.

A few existing timing analysis techniques use assertions of some sort. Flex [22], like Eiffel, is a program with built-in assertions. As a real-time language, these assertions allow run-time checking of timing behavior. The Flex approach is part of the language definition, however, and this does not help with the analysis of Ada.

Ada provides a handy construct for implementing assertions, as well. They are not checked at run-time, however. The pragma statement does not generate code per se. Instead it passes a directive or a suggestion to the compiler on how to compile the code around it. Except for some standard pragmas, pragmas are considered to be implementation defined. Since they may only change the way code is compiled and not its correctness, a compiler must ignore any pragma it does not recognize (although it may print a warning) [23].

The seven assertions defined for this timing analysis are therefore implemented as pragmas. Each is prefixed with "TA_" to help ensure no conflict with any compiler's own pragma set. Pragma statements are very similar to procedure calls. Like procedure calls, arguments to the pragma may be positionally associated or name associated. Either style may be used except with the TA_Time_By_Primitives assertion which must be name associated.

```
TA_Loop_Bound
   (Low, High : natural)
```

This assertion may appear as the first statement in a loop body. If it does, it defines the range of times that the loop will execute. It overrides the bounds derived by analysis of a for-loop specification.

```
TA_Recursion_Bound
   (Low, High : natural)
```

Similar to a loop bound. It may occur as the statement immediately following a self-recursive procedure or function call. Mutual recursion and recursive call chains are not supported.

```
TA_Measure_Start
```

Ignore previous code in this compilation unit for timing analysis purposes. Use this point as an analysis start point. This assertion is intended for use in main programs.

```
TA_Measure_Stop
```

Ignore following code in this compilation unit for timing analysis purposes. This assertion is intended to mark where the end event should be inserted.

```
TA_Time_Absolute
   (Low, High : Natural)
```

This assertion is treated exactly like a code sequence which reduces to an edge with time bounds [low, high] (in cycles).

```
TA_Time_Mix
   (Instruction_Number : Natural;
    Mix : Mix_Type)
```

This assertion is similar to TA_Time_Absolute but uses an instruction count and average instruction time range (mix) to compute the time bounds.

```
TA_Time_By_Primitives
   (<prim_name> => Natural, ...)
```

Like TA_Time_Mix, this assertion takes instruction counts as its arguments. Instead of counting "average" instructions; however, the developer can specify the number of primitive times. Since named association is used, only the primitives of interest need be listed.

Compiler Analysis

In order to determine the values of primitive times, careful analyses of the compiler and the target hardware architecture are required. The compiler analysis must determine the code generated corresponding to each primitive. The hardware architecture analysis must calculate the execution time of this code. Vendor input greatly simplifies the process. However, direct observation of the compiler and hardware may be needed.

The implementation done in conjunction with this thesis uses the Meridian[5] Macintosh Ada compiler operating on a Macintosh IIsi. Neither the compiler nor the hardware were developed with real-time criteria in mind. This means that predictability is not directly supported and that worst-case times may be significantly worse than average case times.

Hardware analysis must consider the instruction timing of the processor along with system interrupt handlers and bus/processor contention for other system maintenance activities. Vendor timing data is crucial for instruction timing. In the case of the Macintosh, its Motorola 68030 processor is described in [24]. Without timing data, extensive testing with logic analyzers would be necessary to measure either instruction timing or primitive routine timing. These tests could not guarantee bounds on these times unless they can guarantee testing all possible conditions for execution. Adequate vendor data utilizes design knowledge to ensure that time bounds given are true bounds or at least bounds under specified conditions. The Motorola data specifies the worst case execution time under assumptions on the length of

---

5 Meridian Ada™ 4.1, Meridian Software Systems Inc., 10 Paseur St., Irvine, CA, 92718.

bus cycles and averaging instruction alignment cases. It does not, however, specify best case execution time.

Code generation analysis is also simplified with access to vendor design data. Particularly, a compiler which uses DIANA as an intermediate representation is relatively easy to trace through the code generation phase relative to the schema. The Meridian compiler, however, does not use DIANA and does not supply insight on code generation. Under these circumstances, analysis may be accomplished by disassembly of the compiler libraries and test programs. These test program listings are compared to the source listing and corresponding DIANA structure to associate primitives with measurable code segments. While this approach lacks the same fundamental guarantees as hardware testing, compiler activity is very likely to follow the constraints of the language definition and common compilation practice. Some of these constraints are embedded in the DIANA construction. This helps make the relevant cases more obvious. In the event of a prediction anomaly, however, a new set of code generation circumstances is one of the first things to look for. The primitive times used in this implementation are developed using the disassembly method described here.

Primitive times disassociate the uniqueness of each compiler/hardware /system grouping, but require analysis of each grouping to determine the values of these primitives. Vendor data greatly simplifies the analysis. A production system would need high quality, high reliability predictions. Vendor data is a fundamental necessity to achieve that level. In the experimental implementation developed with this thesis, vendor data is available on the instruction set timing, but experimentation is used to determine system interference and code generation patterns.

Execution Time Prediction Algorithm

Generate the timing graph as described earlier. Choose start events (nodes) of interest. Select the end event of interest. Sum all edges which "precede" the end event in the graph. An edge "follows" an event when no path can be found beginning with a start event and not including the event. An edge that does not follow an event, precedes it.

Note that without priorities, unbounded loops containing conditional entries and select statements with else clauses create busy tasks. Theoretically, these tasks may run indefinitely without relinquishing the processor. Graphically, unrolling these loops create an indefinite number of edges that have no dependencies on other paths (like a rendezvous does). Thus, if the first of these edges precedes the end event, then the entire unrolled loop can precede the end loop. Thus, these constructs must be used carefully, or else, a method other than the simple graph analysis above must be used (such as the CRSM approach defined in the next chapter).

Discussion

If done by hand, applying this graph construction technique is tedious. It needs to be automated. Non-trivial problems generate extremely large DIANA trees. Automating this turned out to be a difficult problem, however. The difficulties were not in the technique; but instead, in the development environment. The task was larger than the system could handle.

On the other hand, this drove home the need for an automated timing analysis technique. Being forced to use this, relatively abstract, method by hand made me realize the extreme difficulty in evaluating the timing characteristics of the program. It also seemed to show why timing analysis is not done as often as it should.

The other issue with developing this approach is the need for compilers and systems which make some effort to be predictable. The Meridian system used unbounded recursion or iteration in several areas. Unless it was clear that some natural bound applied to the value, this created great difficulty. Alarms, in particular, used several cases of recursion and endless loops — mostly in searching and deleting. For this reason, they could not be adequately characterized so I deferred investigating things like timed_entries.

The combination of assertions, source code analysis and the concurrency analysis upcoming, provides a broad toolkit to the programmer/analyst who needs to track, verify, or bound the performance of his or her system.

Chapter IV

Concurrency Model

Introduction

The amount of concurrency (both real and perceived) completely changes the timing behavior from one concurrency model to the next. Two common models are the *fully concurrent* model and the *interleaved* model. The applicable model is closely tied to system scheduling decisions as discussed earlier. Ada does not specifically require some degree of concurrency or another. Its model is compatible with either a fully concurrent architecture or an interleaved system, as well as combinations of the two.

This thesis applies to the simple case of interleaved concurrency on a single processor. Currently, most Ada compilers are limited to direct exploitation of a single processor using the Ada language constructs. Multiple processors are sometimes made available through usage of underlying operating system capabilities. Besides wishing to avoid incorporating arbitrary operating system characteristics, multiple processors introduce interprocessor communication contention which greatly complicates timing analysis. Some on-going research is directed at the topic of predictable interprocessor scheduling and communication [9, 25-27]. Certainly, the trend is toward multiprocessor systems and direct Ada support of these systems. Extending the approach here to support multiprocessors and resource contention is the logical next step in research.

This chapter discusses Ada's rules on concurrency as well as techniques for modelling Ada concurrency. The two techniques presented are a simple PERT technique and Shaw's Communicating Real-Time State Machines (CRSM) [8]. The PERT technique is the one developed in the current schema definitions. It is usable in single-processor systems with no task priorities. CRSM accounts for prioritization and may be extendable to multiprocessing systems. It is more complex to generate and requires automated support to execute, however. This chapter also outlines an approach for incorporating CRSM into the schema developed here.

Ada Concurrency Model

The Ada Concurrency Model is straightforward. It follows closely Hoare's Communicating Sequential Processes [28, 29]. Tasks embody control flows that may execute in parallel. The Ada main program may be considered as a task for this purpose. At no time may a task with a lower priority run if a task with a higher priority is runable on a given processor. Scheduling decisions between tasks of the same priority is left implementation dependent.

This model implies that a context switch between tasks will only occur when a task blocks or a higher or equal priority task becomes runable. A task may block at any of its *synchronization points*. These include the end of its activation, the activation point of another task, an entry call, the start or the end of an accept statement, a select statement, a delay statement, an exception handler, or an abort statement. It may also block if it uses a blocking system call.[6] In the worst case, a task blocks in all of these situations and a context switch occurs. A task may become runable as a result of an interrupt or expiration of a delay statement. Again, the worst case is that each of these results in a context switch.

For the purposes of this thesis, the model is simplified. Exception handlers and abort statements are ignored as error control statements. While performance under error conditions may also be critical to real-time behavior, the additional complexity which exception handling and aborts introduce requires further work. Furthermore, blocking system calls are not supported since their behavior is implementation dependent. A more general implementation which accounts for resource contention could add this capability. Finally, interrupts are modelled as one class of starting events; that is, as user specified nodes in the timing graph. The time (or relative time) of a series of interrupts must be supplied and the proper entries graph chosen by the user. Interrupts may also be handled solely by the operating system (i.e., clock tick interrupts). In this case the interrupts are not handled by the Ada program. They are ignored in the timing analysis except as they contribute to system interference.

PERT Networks

PERT networks are analyzed by computing the critical path to reach an end event. Parallel tasks are allowed to execute in parallel. That is, PERT

6 In some implementations a blocking system call will block the entire Ada program. This is quite common behavior in Unix where Ada tasks execute as light-weight processes within a Unix process. A system call blocks the entire Unix process.

models full parallelism. Resource levelling is added to PERT analysis to force it not to schedule more parallel activities than resources allow. By specifying that all events use the same resource (CPU) and that there is only one, the PERT analysis begins to model an interleaved system.

The graph supplied for PERT analysis is simply the timing graph with the following modifications. A dummy edge (of zero duration) is added between all dangling activities and the end event. This forces the worst case situation that all events which may possibly precede the end event will do so. Start events must be supplied a time of occurrence. The worst case execution time is then the computed completion time for the end event in whatever time reference was used for specifying the start events. The number of context switching nodes must be counted and multiplied by the worst-case context switch time. This quantity is added to the computed end event time.

This model is limited, however. For instance, PERT cannot handle OR-branching.. OR-branching is when only one graph edge leaving a node is executed in a given instance. This occurs in select and expanded "if or" case statements. With select statements, the choice is driven by the existence of an entry call (or the lack of any). This situation is evident within the structure of the graph. An analyst (or perhaps an automated means) can instantiate the select statements necessary.

There are several nuances with writing rendezvous code. Code can introduce "race" conditions when two tasks call a third which selects between the two call just once. Other difficulties are introduced by rendezvous in a "dependent" context. An independent context is where a select statement can be called an indefinite number of times; or more generally, when the code executed by an entry call is independent of any entry calls by other tasks. If the dependencies in the rendezvous sequence are deterministic, then an analyst may construct the graph in that sequence. When a choice must be made in selecting the caller of particular entry, it may not be evident which choice results in the best or worst time bound. Some choices may not terminate. In the dependent context case, a choice may affect the availability of calls for other choices. In the worst case, all combinations of rendezvous sequences would be tried. Trying all such combinations requires a number of network analyses exponential to the minimum of the number of entry calls and the number of accept points. In almost all cases, programs which may not terminate or which deadlock based on race conditions or which deadlock based on the order of task execution are erroneous.

A simple tasking program illustrates how the PERT technique is used in analyzing timing behavior. The program results in four graphs which must be constructed into a single PERT network. The program is followed by the individual graphs and the resulting network.

```
procedure DOIT is

    COUNT_A, COUNT_B : INTEGER := 0;
    RESULT_C         : INTEGER := 6;

    task TASK_A is
    end TASK_A;

    task TASK_B is
    end TASK_B;

    task TASK_C is
        entry ENTRY_A;
        entry ENTRY_B;
        entry DONE;
    end TASK_C;

    task body TASK_A is
    begin
        for I in 1 .. 1000 loop
            COUNT_A := COUNT_A + 1;
        end loop;
        TASK_C.ENTRY_A;
        COUNT_A := COUNT_A + COUNT_A;
    end TASK_A;

    task body TASK_B is
    begin
        for I in 1 .. 100 loop
            COUNT_B := COUNT_B + I;
        end loop;
        TASK_C.ENTRY_B;
        COUNT_B := COUNT_B + COUNT_B;
    end TASK_B;

    task body TASK_C is
    begin
        for I in 1 .. 2 loop
            select
               accept ENTRY_A do
                  RESULT_C := RESULT_C / 2;
               end ENTRY_A;
            or
               accept ENTRY_B do
                  RESULT_C := RESULT_C + 4;
               end ENTRY_B;
            end select;
        end loop;
        accept DONE;
    end TASK_C;

begin -- DOIT
    TASK_C.DONE;
end DOIT;
```

Figure 6: Simple Tasking Program

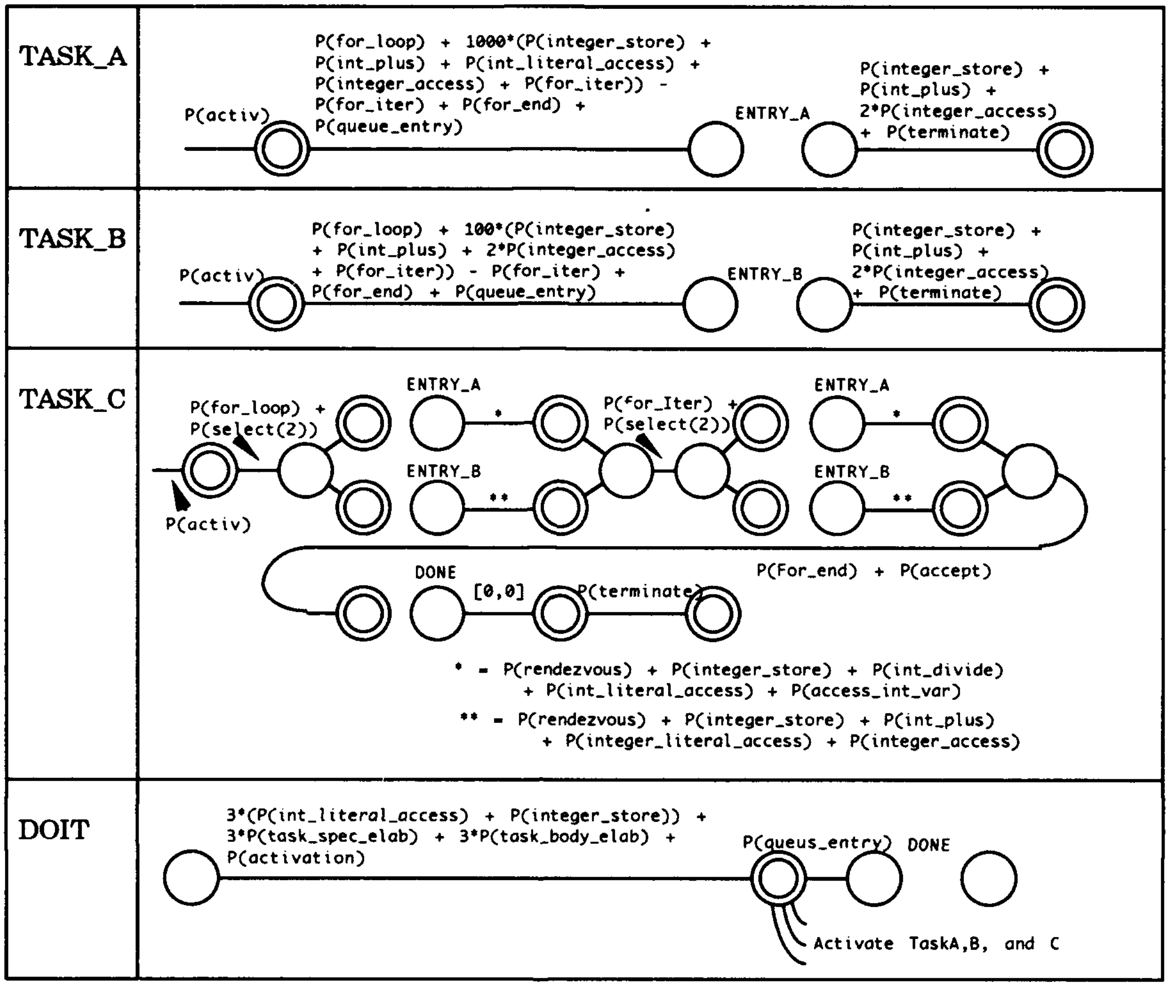


Figure 7: Timing Graphs for Simple Tasking Program

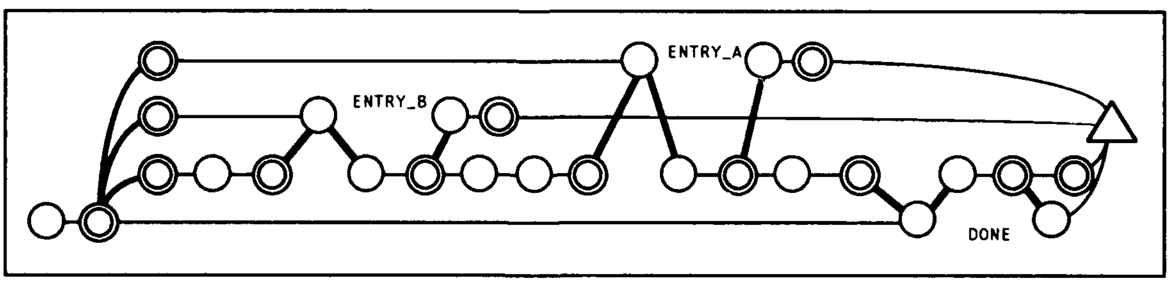


Figure 8: PERT Network for Simple Tasking Program

In figure 8, the PERT network is simply constructed by matching together the timing graphs for the various tasks and subprograms. The boldface edges are the ones added to construct the network. In this case, they are added at entry calls and task activations. The triangular node represents the end event. Zero-weighted edges tie the completion of all relevant threads to this event. Note that the entry calls in Task A and B are independent. The network could be constructed with the accepts in Task C reversed. The same edges and nodes would still precede the end event; the order does not matter.

Communicating Real-Time State Machines

Communicating Real-Time State Machines (CRSM) are an executable specification technique [8]. Their key feature is the capability for describing timing properties. They are described by a system model and operational semantics. Each concurrent task is represented by a state machine with synchronous intertask communication. The following paragraphs briefly describe what is developed in the above citation.

The system model is a set of state machines and communication *channels*. The state machines are described by a set of states and transitions. The transitions have labels of the form *guard* → *command*. Guards are conditions that must be satisfied before execution of the associated command. Omitting the guard is equivalent to a guard of *true*. The commands may change local variables and/or communicate with other machines. Channels abstractly represent communication between two machines. They are identified by an event name. The event may have parameters associated with it; these are set during the communication.

The operational semantics describe how to transform an input of CRSM's and their channels into a time-sequenced event trace. The basic approach is to construct next event lists for each machine. This is followed by selecting the earliest event(s) from the various lists. These are executed, time updated, and the process repeated.

Time is represented as a range [min, max] associated with each transition. This is the time it takes for the transition to execute. Communication occurs instantaneously. Also each machine has an associated real-time machine which can be used to model delays as well as get time stamps.

CRSM and Ada Tasking Structures

The basic mapping between Ada and CRSM is to model each Ada task (and master subprogram) as a machine. Transitions represent execution of some statement or sequence of statements. Communication between tasks (entries) are modelled using communication channels. The main change is that channels can have *out* variables passed as actual parameters. The event synchronization does not map exactly to an Ada rendezvous which synchronizes the sender and receiver for some bounded but significant amount of time. Thus, events are used to synchronize both the start and the end of the rendezvous. Furthermore, the calling machine is not allowed any other transitions between the events starting and ending the rendezvous.

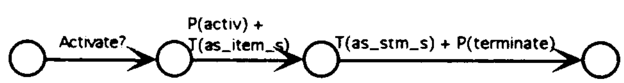
Operationally, the timing analysis technique models programs on a single processor. Therefore, CRSMs are executed as described, but instead of running all machines in parallel, only one machine is selected to run at at time. A trace begins with the event of interest and an arbitrary runable machine is selected to run at each "blocking" point. Because of Ada interleave semantics, the order transitions are executed is not important (for independent calling contexts). Priorities can be introduced by making scheduling decisions based on the Ada priority scheme.

Because CRSM and Ada both have roots in Hoare's CSP, it is not surprising that the mapping between them is straightforward. By reintroducing parallelism to the operational semantics, CRSM can simulate multi-processing of Ada programs. This and the ability to make scheduling decisions give CRSM a great deal more power than a simple PERT representation of the program.

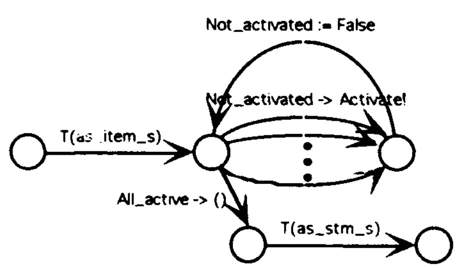
Integrating Schema Analysis and CRSM Construction

Constructing timing models using CRSM is not much different than using PERT. Both techniques use graphs. The difference is in the forms of the graph; and, of course, what they mean and how they are analyzed. Thus, the same schema apply except for some of those which construct graph elements. These schema are replaced by the constructs in the following table. Input events are marked with question marks. These correspond to output events with the same name and marked with exclamation points. The input RT?[x] is an input from the real-time machine associated with the task to occur at least x seconds in the future.

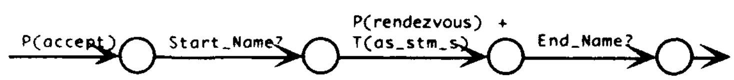
Task_Body



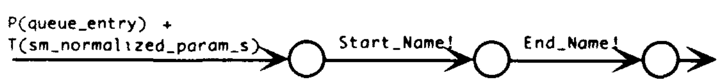
Enclosing Block

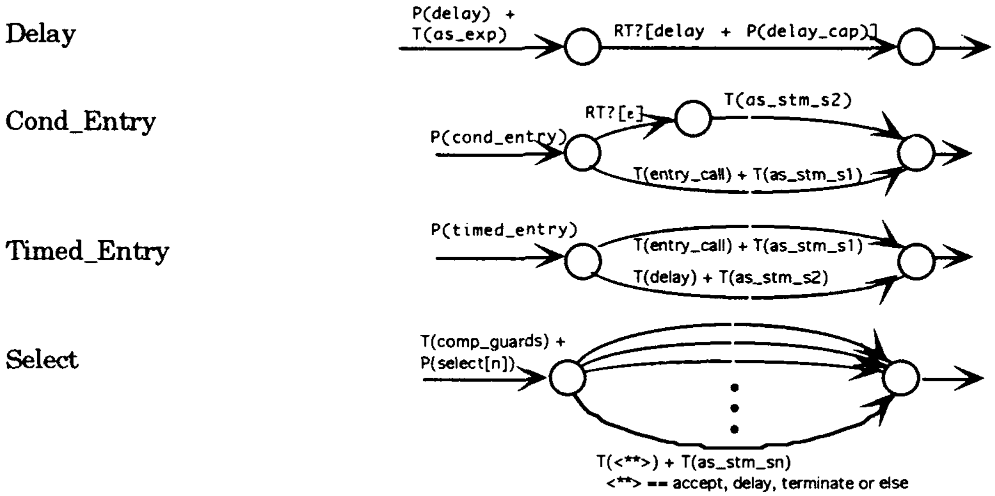


Accept



Entry_Call





For the worst case analysis, the max time is used for each transition. For the best case analysis, the min time is used. The completion of the analysis occurs when the activities triggered by the starting events subside. Note that this model does not require assembly of the generated timing graphs. They are input to the operational CRSM model as is.

Dependent tasking contexts introduce similar problems in this model since the calling order may change based on the length of execution at some point. This change in calling order may result in an overall faster or slower time. Thus, always choosing the max or min time may not generate the worst or best case respectively.

Discussion

The concurrent constructs of Ada require special consideration for timing analysis. On a single processor, only one task can use the processor at a time. This interleaved model means that the execution time of a program is the sum of the time spent in each task with one exception. If any tasks use delays, then it may be possible that no task is runable at some point. This idle time must also be added to the program execution time. PERT analysis can help make this determination.

CRSM provides a more powerful model. Although developed as a specification method, it serves well as a descriptive technique. It can model multiple processors as well as more complex scheduling decisions. Its drawback is the requirement for a specific tool to run the model.

Dependent tasking contexts are system designs where the timing properties are dependent on the order that entry calls occur. Neither PERT nor CRSM can simply determine the bounded execution time of such programs. Trying all possible orders of entry calls is the sure way of finding the bounded execution time; however, this is an exponential growth approach. Further understanding and characterizing the conditions which cause dependent tasking contexts is necessary to determine if a better approach can be developed or if the class of programs can be ruled out.

Chapter V

Experiments

Introduction

Experiments consist of benchmark tests. These tests are compiled and run in a *dedicated* system environment. They are also converted to a DIANA representation which is fed to an automating timing graph generator. The resulting timing graphs are combined into a network and analyzed by hand. The results are adjusted for system interference are compared to the experimentally observed times. In a few cases, both the results and experimental times are compared to worst case times computed using hand analysis techniques like Knuth's.

Setup of the Experiments

As stated in the introduction, the experiments consist of Ada programs which undergo timing analysis, which are timed while executed, and whose prediction and execution results are compared. The benchmark programs come from two sources. The first is the Special Interest Group for Ada of the

Association for Computing Machinery (ACM SIGAda) Performance Issues Working Group (PIWG). The second are programs specifically constructed to include a wider selection of language features and exercise some of the capabilities of the analyzer.

The PIWG has constructed a series of benchmarks that measure and compare various features of the Ada language. The benchmarks measure both compilation and execution performance. For these experiments, the best choices are execution benchmarks which test basic sequential language constructs and which test simple tasking situations. Many of these benchmarks are relatively simple, so their object code modules may also be hand analyzed for timing behavior.

The other set of programs are somewhat more complex, but still relatively small (less than 100 SLOC). In both cases the programs are enclosed within an iteration loop. The computer clock is read immediately before the iteration loop and immediately after it completes. The iteration loop may be run several times. In the case of the PIWG benchmarks, the number of iterations varies from one run to the next.

Each program is compiled on a Rational R1000.[7] The Rational creates a DIANA tree for the program. The DIANA tree is copied into a text file and transferred back to the Macintosh. The Rational represents node identifiers with a long hexadecimal value; pointers in semantic attributes are marked with a caret (^). These values are replaced with a simple integer from the set 1 to n where n is the number of nodes in the file. Semantic pointers are replaced with a similar value with a package id extension (e.g., ^standard.9). If it had been available, this file would be loaded into the timing analysis

---

7 Rational and R1000 are registered trademarks of Rational, 1501 Salado Drive, Mountain View, California, 94043.

program which generates timing graphs for its various program units (i.e., subprograms, tasks, package elaborations). Note that compilation units upon which it depends must be loaded before it; particularly the standard package. The timing graphs which result are printed. Without the analysis program, the timing graphs are generated by hand application of the schema to the DIANA representations. Building networks from these graphs and analyzing them generate the bounded execution time predictions of interest.

The expected execution time is then compared to actual executions. The program is compiled and run on the Macintosh system. No other application programs are run including system extensions like screen savers or virus protection software. Furthermore, keys are not pressed after beginning the test execution and the mouse is not moved. Other programs, particularly system extensions, and input device activity all add to system interference. The programs do not require disk or screen I/O within the critical timing section. The benchmark programs complete by printing out the timing measurements it made.

The execution time of the program (T_{ex}) is represented in the equation:

$$T_{measured} = T_{clock} + T_{loop\text{-}overhead} + I \times T_{ex}$$

where I is the number of iterations in the loop. Solving this for T_{ex} results in

$$T_{ex} = [T_{measured} - (T_{clock} + T_{loop\text{-}overhead})] / I \approx T_{measured} / I$$

if I is large enough. Before comparison, the predicted times must be adjusted for the estimated system interference experienced by the test.

System interference is the amount of time the system spends handling interrupts rather than running the program of interest. A simple benchmark measures the amount of time in a simple loop. Running this benchmark under the system configuration described above gives the data necessary to derive a range of nominal system interference. The benchmark was disassembled and hand analyzed. The result was compared to the timed execution result and a range for system interference determined.

A better experiment was attempted. This would measure the execution time with interrupts disabled and compare it to the normally measured time. The relative difference would represent system interference. The experiment failed because the system routine for reading the hardware clock directly did not work and the other clocks were interrupt driven. Note that disabling interrupts for any significant period of time, in general, would break the Ada run-time system and thus is not feasible for application experiments . However, until the problems mentioned above can be fixed, disabling interrupts is not viable for characterizing system interference, either.

The expected execution time of the system interference benchmark was computed as requiring 228,030,018 - 250,035,023 cycles which at 25 MHz equals 9.12 - 10.00 seconds. The measurement was repeated 100 times. Each measurement was either 14 or 15 seconds. The first time was observed 45 times; thus the weighted average is 14.55 seconds. This means the time spent in interrupt handlers is 4.55 - 5.43 seconds distributed across 60 ticks per second. From this data the average time spent in the tick interrupt handler is 5.21 - 6.22 ms. The nominal system interference, the percentage of each tick spent handling interrupts, is then 31.3 - 37.3%.[8] Thus, 62.7 - 68.7% of the processor is available for the experiment and the predicted times should be divided by these amounts to give the comparable predictions with system interference accounted for.

---

8 With five system extensions installed, the system interference rises to 60-65%.

Experimental Results

The following table summarizes the results collected. All results are tabulated in Appendix D. Without the automation to generate timing graphs, very few predictions were completed. The predicted times are corrected for system interference.

PIWG Experiment	256 Iterations Measured	1024 Iterations Measured	8192 Iterations Measured	256 Iterations Predicted	1024 Iterations Predicted	8192 Iterations Predicted
C000001 T	27 - 27	108 - 108	n/m			
C000002 T	28 - 29	113 - 113	n/m			
H000004 T	13 - 13	52 - 52	n/m			
P000001 C	1 - 2	5 - 6	41 - 48	1.17 - 1.58	4.67 - 6.31	37.4 - 50.5
P000001 T	1 - 2	5 - 6	48 - 52	1.27 - 2.01	5.08 - 8.02	40.6 - 64.2
P000010 T	2 - 3	8 - 8	69 - 70			
T000001 T	13 - 13	51 - 52	n/m			
T000004 T	30 - 30	119 - 121	n/m			

Table 1: Iterated Experiment Results

Experiment	Measured Time	Predicted Time
A000091	1.10 - 1.20 ms	
A000092	2.78 - 2.84 ms	
SimpleTasks	11.8 - 14.6	

Table 2: Single result experiments

Interpretation

For the experiments completed, the predictions bound the measurements (as shown in figures 9 and 10). This is hardly surprising if the primitives and schema are defined and calculated correctly. The question is then how good the bounds are. The limiting condition on the tightness of the bounds is the tightness of the primitives involved in the prediction.

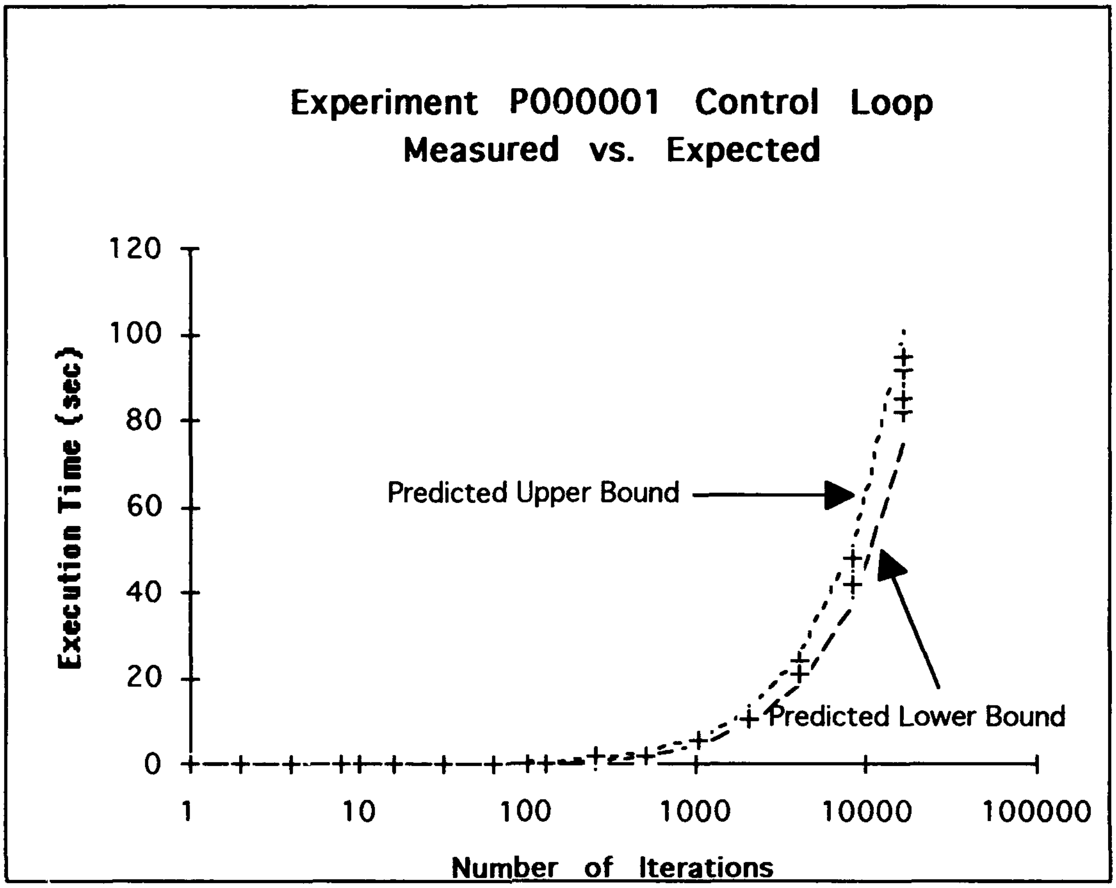


Figure 9: A Plot of Experiment P000001 Control Loop Results and Expectations.

The upper bound of each primitive can be expressed as the lower bound multiplied by a factor. The lower bound prediction is the sum of the primitive lower bounds. The upper bound prediction can be expressed as the lower bound multiplied by a weighted average of the primitive upper bound factors. The average is weighted by the frequency of occurrence for a given primitive in the prediction and by the amount of time represented by the primitive.

Some interesting limits arise from this view. Foremost is the observation that the difference between the actual program execution time and the upper bound must be less than the factor between the lower and upper bounds. In the P000001 experiments above, the factor between the bounds are 1.35 for the control loop and 1.58 for the test loop.

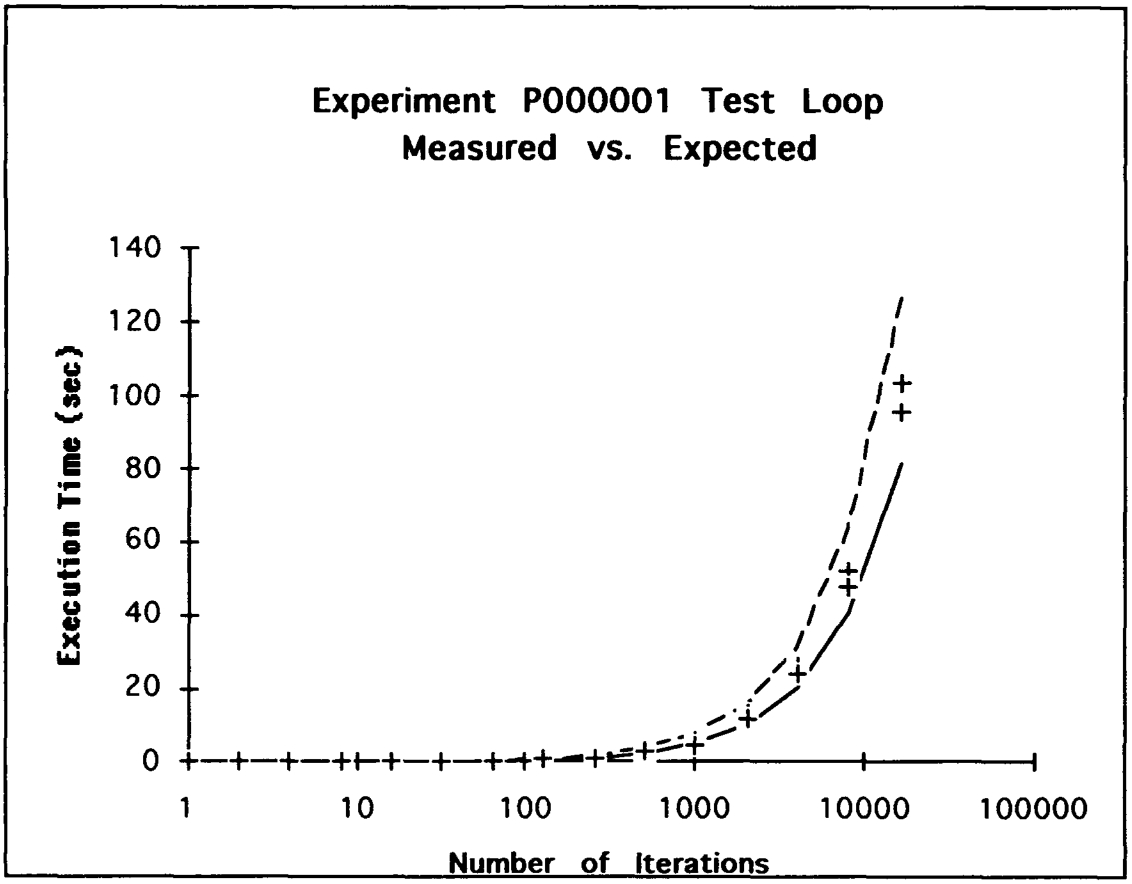


Figure 10: A Plot of Experiment P000001 Test Loop Results and Expectations.

The second observation is that the factor for the code in a loop body is invariant, i.e., it does not change if the number of loop iterations change. Plotting the upper and lower bounds against iterations of the loop on log-log charts would generate parallel lines. This is the case, in fact, for the graphs above when plotted on a log-log rather than semi-log scale.

The final observation applies when branching and variable loop bounds are ignored. The factor between the bounds of the prediction cannot exceed the largest factor between the bounds of any primitive used in the calculation. This follows from the way a weighted average works. An average cannot exceed the largest of its input data. On the a"chitecture studied, many primitives only vary by 30 - 50%, others vary by factors of four or more. Tight primitive bounds result in tight prediction bounds.

In the experiments above, the worst case bound exceeded the worst case measurement by factors of 5.2% and 33.7% for the control loop and test loop respectively. However, testing may not execute the worst case. With the low number of runs conducted so far these numbers, particularly the second, are probably high. The observation in the previous paragraph, however, indicates that the P000001 code is relatively tight compared to the primitives that may be encountered in other code. Examples involving tasking, delays, or floating point arithmetic have much larger factors between their bounds.

Coverage of the execution results offer another perspective on how good the predicted bounds are. Coverage is the range of test results from the lowest time to the highest time compared to the range of the bounds. For instance, the measurement runs for the P000001 control loop cover iterated 8192 times cover the range 41 - 48 seconds. The bounds are 37.4 and 50.5 seconds. So, the results cover seven seconds of the 13.1 seconds between the bounds or 53%.

Without more testing and study of the implications of these metrics, it is imprudent to draw any conclusions. The metrics of difference factors and coverage help visualize how the predictions stack up to the actual executions. Predictions that generate a factor of 1 between worst case execution and the worst case bound and 100% coverage are obviously ideal. They may be unrealistic; but may also, in fact, be achievable by applying the technique to systems with hardware and compilers developed to generate predictable code.

Chapter VI

Discussion and Future Research

The proposed approach to timing analysis is promising. It satisfies the objectives outlined in Chapter I, but can certainly be improved. Specifically, it does not take advantage of all the context information available to it; it does not handle exception processing or I/O; and it is limited to single processor systems. These are all areas that should be pursued further. Additionally, continuing improvement can be made to tighten the bounds by tightening the primitive time bounds and manipulating the schema to fit better.

In extending this approach it is important to keep the objectives in mind. A particularly difficult one is portability, the applicability of the technique to different hardware and compiler systems. By manipulating the schema to fit more closely what a particular compiler does, the performance with that compiler will improve. It may be incompatible, however, with another compiler. Thus, the schema must continue to conform closely with the language definition.

Park and Shaw has already somewhat considered the tradeoff between tightness of the execution time bounds and portability [30]. They look at large and small atomic blocks represented by their primitive times. This is another way of factoring context into the picture. By defining primitives in larger terms (such as an entire assignment statement), one can better characterize the compilers behavior. The disadvantage is that many more primitives are needed (a single assignment primitive is not sufficient; one will be needed for each major type and for significant optimization patterns). The goal is to determine a balanced set of primitives and the method for identifying the proper context for using them.

The most difficult part of the language and most implementations is the tasking model and constructs. Better characterization and understanding of tasking activity is a high priority. Here more than any place else, support by the compiler vendor is crucial to understanding the system. Tasking implementations are 10,000's of lines of code. The exercise of characterizing the tasking behavior of the compiler may also benefit the compiler vendor who should discover what parts of the implementation are least predictable and most difficult to characterize.

Another way to distinguish characterizations is using *modes* and *variability*. I use these terms to describe the things which vary execution times from one system or instance to the next. Modes are those things that are fixed at some point in instantiation of the system (e.g., instruction word alignment in the object code image). Variability refers to environmental factors which may continue to vary (e.g., input values, competing workloads, and operator controls). By so categorizing these things, further characterization of the system may become available as modes are fixed for a particular instance.

Modes and variability can be modelled to some degree with random processes and variables. A more general understanding of the system is potentially available using probabilistic models like those in [31] and [32]. Characterizing primitive times as random variables may allow further a priori characterization of a prediction's factors and ranges. An important distinction that may be provided by studying modes and variability is separation of dependent and independent random variables. These models may allow for reasonable specification and verification of a non-perfect system, e.g., the system that requires 98% availability.

The trickiest problem that must be addressed is multi-processing and shared resources. Gerber and others have started looking at timing analysis in light of resource contention [33]. Tying resource sharing into a general abstract timing analysis model would provide a critical tool to the developers of real-time systems.

I intend to continue exploring this subject as my research area at the US Air Force Academy. Based on personal experience and the literature, systems developers need timing analysis techniques that can apply to the systems they're building today and will be building tomorrow. These techniques must span the entire development cycle, be general enough to use on several projects, and provide reliable performance.

Appendices

Appendix A

Timing Primitives for Mac IIsi and Meridian Ada

The following lists the timing primitives as identified by analysis of the Ada language, DIANA representation and Meridian Ada code generation. Where a high or low execution timing bound has been determined for the primitive, the value is shown in cycles. The Mac IIsi executes at 25 MHz.

accept		
access		
activ	184	618
activation		
allocate_main		
and_then		
array_access		
array_aggr_comp_access		
array_aggr_setup		
array_cat		
array_comp_access		
array_comp_store		
array_ge		
array_greater		
array_in		
array_le		
array_lesser		
array_not_in		
array_store		
attr_address		
attr_aft		
attr_base		
attr_callable		
attr_constrained		
attr_count		
attr_delta		
attr_digits		
attr_emax		
attr_epsilon		
attr_first		
attr_first		
attr_first(N)		
attr_first_bit		
attr_fore		
attr_image		
attr_large		
attr_last		
attr_last(N)		
attr_last_bit		
attr_length		
attr_length(N)		
attr_machine_emax		
attr_machine_emin		
attr_machine_mantissa		
attr_machine_overflows		
attr_machine_radix		
attr_machine_rounds		
attr_mantissa		

attr_pos		
attr_position		
attr_pred		
attr_range		
attr_range(N)		
attr_safe_emax		
attr_safe_large		
attr_safe_small		
attr_size		
attr_small		
attr_storage_size		
attr_succ		
attr_terminated		
attr_val		
attr_value		
attr_width		
ba_and		
ba_not		
ba_or		
ba_xor		
bool_access		
bool_and		
bool_eq		
bool_neg		
bool_not		
bool_or		
bool_store		
bool_xor		
cond_entry		
case		
context switch		
convert_array		
convert_derived		
convert_fixed2float		
convert_fixed2int		
convert_float2fixed		
convert_float2int		
convert_int2fixed		
convert_int2float		
default_param		
delay		
else		
fixed_abs		
fixed_access		
fixed_div		
fixed_eq		
fixed_ge		
fixed_greater		
fixed_identity		
fixed_in		
fixed_le		
fixed_lesser		
fixed_minus		
fixed_mul		
fixed_neq		
fixed_negation		
fixed_not_in		
fixed_plus		
fixed_store		
float_access		
float_abs		
float_div		
float_eq		
float_exp		
float_identity		
float_in		
float_minus		
float_mul		
float_neq		
float_negation		
float_not_in		
float_plus		
float_store		
for_end	14	22
for_iter	34	50
for_loop	6	36
function_access	0	0
function_call		
function_end		
if		
int_abs		
int_div		
int_eq		
int_exp		
int_ge		
int_greater		
int_identity		
int_in		
int_le		
int_lesser		
int_literal_access	0	6
int_minus		
int_mod		
int_neq		

int_negation		
int_not_in		
int_plus	8	12
int_rem		
int_times		
integer_access	4	16
integer_store	5	18
iter		
loop		
loop_end		
null		
or_else		
package_elaboration		
procedure_call [9]	40	858
procedure_end	28	100
queue_entry	992	2082
range_check		
real_literal_access		
record		
record_aggr_comp_acces		
record_aggr_setup		
record_comp_access		
record_comp_store		
rendevous		
select		
slice_access		
slice_store		
string_literal_access		
task_body_elab	92	158
task_spec_elab	2202	4166
terminate	24	94
timed_entry		
variant_tag_check		

9 Worst case time is only 301 when the procedure contains no tasks.

Appendix B

Test Program Source Code

A000001

```
with TEXT_IO ; use TEXT_IO ;

package DURATION_IO is new FIXED_IO ( DURATION ) ;
```

A000018

```
-- This is a universal Ada function to get CPU time in seconds
-- of type DURATION on non time_sharring systems where a
-- tailored CPU_TIME_CLOCK is not reasonable
-- Do not cross a midnight boundry

-- It is modified to read the clock using the Mac OS clock routine rather
--  than the calendar package.  This gives 1/60th second rather than 1 sec
--  resolution.

with EVENTS;
with MAC_TYPES;

function CPU_TIME_CLOCK return DURATION is

        MaxTicks : Constant := 60 * 86400; -- Duration'last
        NOW : MAC_TYPES.LONGINT := EVENTS.TICKCOUNT ;
begin

  return DURATION ( FLOAT (NOW mod maxTicks) / 60.0 ) ;

end CPU_TIME_CLOCK ;
```

A000021

```
package REMOTE_GLOBAL is  -- for explicit control of optimization

  A_ONE : INTEGER; --        a constant 1 that can not be optimized away
--                           A_ONE is intentionally visible. DO NOT CHANGE IT
--
  GLOBAL : INTEGER := 1 ; -- global object can not be optimized away
--                           GLOBAL is changed by measurement programs
--                           the initialization to 1 is used in the body
--                           but could be changed by elaboration order
--
  procedure REMOTE; --       do to calls to this procedure, no compiler
--                           can optimize away the computation an GLOBAL
--
  procedure CHECK_TIME ( TEST_DURATION : in DURATION ) ;
--  Just print message if TEST_DURATION less then
--                           100 * SYSTEM.TICK  or DURATION'SMALL
--
end REMOTE_GLOBAL;
```

A000022

```
with SYSTEM, TEXT_IO ;

package body REMOTE_GLOBAL is  -- must be compiled last
--                                for explicit control of optimization
  LOCAL : INTEGER;

  procedure REMOTE is -- this is an optimization control procedure
  begin
    GLOBAL := GLOBAL + LOCAL; -- be sure procedure is not optimized away
  exception
    when NUMERIC_ERROR =>
      REMOTE ; -- can not happen if test is working ( prevents inlining )
  end REMOTE;

  procedure CHECK_TIME ( TEST_DURATION : in DURATION ) is
  begin
    if TEST_DURATION < 100 * DURATION'SMALL or
       TEST_DURATION < 100 * SYSTEM.TICK then
      TEXT_IO.PUT_LINE ( " ***** TEST_DURATION not large compared to "
                       & "DURATION'SMALL or SYSTEM.TICK " ) ;
    end if ;
  end CHECK_TIME ;

begin

  A_ONE := 1 ; -- must not be changed by measurement programs
  LOCAL := GLOBAL - A_ONE; -- really a zero but compiler doesn't know

end REMOTE_GLOBAL;
-- This is the ITERATION_COUNT control package for feature measurements
-- The set of procedures provide the automatic stabilizing of the
-- timing measurement. The measurement CPU time must be greater than:
--   1.0 second,  DURATION'SMALL * 100 ,  SYSTEM.TICK * 100
--
-- Note: If there is no control loop, the START_CONTROL and STOP_CONTROL
--       do not need to be called.
```

```
package ITERATION is -- A000031.ADA

  subtype ITERATION_COUNTS is INTEGER range 1 .. 32768;

  procedure START_CONTROL ;

  procedure STOP_CONTROL ( GLOBAL : INTEGER ;
                           CHECK : INTEGER ) ;

  procedure START_TEST ;

  procedure STOP_TEST ( GLOBAL : INTEGER ;
                        CHECK : INTEGER ) ;

  procedure FEATURE_TIMES ( CPU_TIME : out DURATION ;
                            WALL_TIME : out DURATION ) ;

  procedure INITIALIZE ( ITERATION_COUNT : out INTEGER ) ;

  procedure TEST_STABLE ( ITERATION_COUNT : in out INTEGER ;
                          STABLE : out BOOLEAN ) ;
end ITERATION ;
```

A000033

```
-- Iteration control package body ( for test development )
--   This version is instrumented and may interefere with some
--   types of tests

with CPU_TIME_CLOCK ;  -- various choices on tape
with CALENDAR ; -- used for WALL clock times
with SYSTEM ; -- used to get value of TICK
with TEXT_IO ;  -- only for diagnostics
with DURATION_IO ;

package body ITERATION is -- A000032.ADA

--
-- CPU time variables
--
  CONTROL_TIME_INITIAL : DURATION ; -- sampled from CPU_TIME_CLOCK at beginning
  CONTROL_TIME_FINAL : DURATION ;  -- sampled from CPU_TIME_CLOCK at end
  CONTROL_DURATION : DURATION ; -- (FINAL-INITIAL) the measured time in seconds
  TEST_TIME_INITIAL : DURATION ; -- ditto for TEST
  TEST_TIME_FINAL : DURATION ;
  TEST_DURATION : DURATION ;
--
-- WALL time variables
--
  WALL_CONTROL_TIME_INITIAL : DURATION ; -- sampled from CLOCK at beginning
  WALL_CONTROL_TIME_FINAL : DURATION ;  -- sampled from CLOCK at end
  WALL_CONTROL_DURATION : DURATION ; -- (FINAL-INITIAL) measured time in seconds
  WALL_TEST_TIME_INITIAL : DURATION ; -- ditto for TEST
  WALL_TEST_TIME_FINAL : DURATION ;
  WALL_TEST_DURATION : DURATION ;
--
  MINIMUM_TIME : DURATION := 1.0 ; -- required minimum value of test time
  TEMP_TIME : FLOAT ; -- for scaling to microseconds
  ITERATION_COUNT : ITERATION_COUNTS ;  -- change to make timing stable
  CHECK : INTEGER ; -- saved from STOP_TEST call for scaling

  procedure START_CONTROL is
  begin
    CONTROL_TIME_INITIAL := CPU_TIME_CLOCK ;
    WALL_CONTROL_TIME_INITIAL := CALENDAR.SECONDS(CALENDAR.CLOCK) ;
  end START_CONTROL ;

  procedure STOP_CONTROL ( GLOBAL : INTEGER ;
                           CHECK : INTEGER ) is
  begin
    CONTROL_TIME_FINAL := CPU_TIME_CLOCK ;
    CONTROL_DURATION := CONTROL_TIME_FINAL - CONTROL_TIME_INITIAL ;
    WALL_CONTROL_TIME_FINAL := CALENDAR.SECONDS(CALENDAR.CLOCK) ;
    WALL_CONTROL_DURATION := WALL_CONTROL_TIME_FINAL -
                             WALL_CONTROL_TIME_INITIAL ;
--
    if CHECK /= GLOBAL then
      TEXT_IO.PUT_LINE ( " Fix control loop before making measurements." ) ;
      TEXT_IO.PUT_LINE ( INTEGER'IMAGE ( GLOBAL ) & " = GLOBAL " ) ;
      raise PROGRAM_ERROR ;
    end if ;
    TEXT_IO.PUT_LINE ( "Iteration " & INTEGER'IMAGE ( ITERATION_COUNT ) ) ;
    DURATION_IO.PUT ( CONTROL_TIME_INITIAL );
    DURATION_IO.PUT ( CONTROL_TIME_FINAL );
    DURATION_IO.PUT ( CONTROL_DURATION );
    TEXT_IO.NEW_LINE ;
  end STOP_CONTROL ;

  procedure START_TEST is
  begin
    TEST_TIME_INITIAL := CPU_TIME_CLOCK ;
    WALL_TEST_TIME_INITIAL := CALENDAR.SECONDS(CALENDAR.CLOCK) ;
  end START_TEST ;

  procedure STOP_TEST ( GLOBAL : INTEGER ;
                        CHECK : INTEGER ) is
  begin
```

```
    TEST_TIME_FINAL := CPU_TIME_CLOCK ;
    TEST_DURATION := TEST_TIME_FINAL - TEST_TIME_INITIAL ;
    WALL_TEST_TIME_FINAL := CALENDAR.SECONDS(CALENDAR.CLOCK) ;
    WALL_TEST_DURATION := WALL_TEST_TIME_FINAL - WALL_TEST_TIME_INITIAL ;
--
    ITERATION.CHECK := CHECK ;
    if CHECK /= GLOBAL then
      TEXT_IO.PUT_LINE ( " Fix test loop before making measurements." ) ;
      TEXT_IO.PUT_LINE ( INTEGER'IMAGE ( GLOBAL ) & " = GLOBAL " ) ;
      raise PROGRAM_ERROR ;
    end if ;
  end STOP_TEST ;

  procedure FEATURE_TIMES ( CPU_TIME : out DURATION ;
                            WALL_TIME : out DURATION ) is
  begin
--
--  compute scaled results
--
    begin
      TEMP_TIME := FLOAT ( TEST_DURATION - CONTROL_DURATION ) ;
      TEMP_TIME := (1_000_000.0 * TEMP_TIME) /
                   ( FLOAT ( ITERATION_COUNT ) * FLOAT (CHECK) );
      CPU_TIME := DURATION ( TEMP_TIME ) ;
    exception
      when others =>  -- bail out if trouble in conversion
        CPU_TIME := 0.0 ;
    end ;
--
    begin
      TEMP_TIME := FLOAT ( WALL_TEST_DURATION - WALL_CONTROL_DURATION ) ;
      TEMP_TIME := (1_000_000.0 * TEMP_TIME) /
                   ( FLOAT ( ITERATION_COUNT ) * FLOAT (CHECK) );
      WALL_TIME := DURATION ( TEMP_TIME ) ;
    exception
      when others =>
        WALL_TIME := 0.0 ;
    end ;

  end FEATURE_TIMES ;


  procedure INITIALIZE ( ITERATION_COUNT : out INTEGER ) is
  begin
    ITERATION_COUNT := 1 ;
    ITERATION.ITERATION_COUNT := 1 ;
  end INITIALIZE ;

  procedure TEST_STABLE ( ITERATION_COUNT : in out INTEGER ;
                          STABLE : out BOOLEAN ) is
  begin
    if TEST_DURATION > MINIMUM_TIME then
      STABLE := TRUE ;
    elsif ITERATION_COUNT >= 16384 then
      TEXT_IO.PUT_LINE ( "***** INCOMPLETE MEASUREMENT *****" ) ;
      STABLE := TRUE ;
    else
      ITERATION_COUNT := ITERATION_COUNT + ITERATION_COUNT ;
      ITERATION.ITERATION_COUNT := ITERATION_COUNT ;
      STABLE := FALSE ;
    END IF;
  end TEST_STABLE ;


--
begin

  if SYSTEM.TICK * 100 > MINIMUM_TIME then
    MINIMUM_TIME := SYSTEM.TICK * 100 ;
  end if;

  if DURATION'SMALL * 100 > MINIMUM_TIME then
    MINIMUM_TIME := DURATION'SMALL * 100 ;
```

```
  end if;

-- MINIMUM_TIME is now the larger of 1.0 second,
--                                   100*SYSTEM.TICK,
--                                   100*DURATION'SMALL

  CONTROL_DURATION := 0.0 ;
  WALL_CONTROL_DURATION := 0.0 ;

end ITERATION ;
```

A000091

```
---------------------------------------------------------------------------
--                                                                       --
--               "DHRYSTONE" Benchmark Program                           --
--               -----------------------------                           --
--                                                                       --
--        Version ADA/1                                                  --
--                                                                       --
--        Date:   04/15/84                                               --
--                                                                       --
--        Author: Reinhold P. Weicker                                    --
--                                                                       --
--                                                                       --
--  As published in Communications of ACM, October 1984  Vol 27 No 10    --
--                                                                       --
---------------------------------------------------------------------------
--                                                                       --
-- The following program contains statements of a high-level programming --
-- language (Ada) in a distribution considered representative:           --
--                                                                       --
--   assignments               53%                                       --
--   control statements        32%                                       --
--   procedures, function call 15%                                       --

-- 100 statements are dynamically executed. The program is balanced with --
-- respect to the three aspects:                                         --
--                                                                       --
--     - statement type                                                  --
--     - operand type (for simple data types)                            --
--     - operand access                                                  --
--          operand global, local, parameter, or constant.               --
--                                                                       --
-- The combination of these three aspects is balanced only approximately. --
--                                                                       --
-- The program does not compute anything meaningful, but it is syntactically --
-- and semantically correct. All variables have a value assigned to them --
-- before they are used as a source operand                              --
---------------------------------------------------------------------------
package global_def is
------------------

-- global definintions

type Enumeration is (ident_1,ident_2,ident_3,ident_4,ident_5);

subtype one_to_thirty is integer range 1..30;
subtype one_to_fifty is integer range 1..50;
subtype capital_letter is character range 'A'..'Z';

type String_30 is array(one_to_thirty) of character;
    pragma pack(string_30);

type array_1_dim_integer is array (one_to_fifty) of integer;
type array_2_dim_integer is array (one_to_fifty,
                                   one_to_fifty)  of integer;

type record_type(discr:enumeration:=ident_1);

type record_pointer is access record_type;

type record_type(discr:enumeration:=ident_1) is
    record
        pointer_comp:           record_pointer;
        case discr is
        when ident_1 =>         -- only this variant is used,
                                -- but in some cases discriminant
                                -- checks are necessary
          enum_comp:            enumeration;
          int_comp:             one_to_fifty;
          string_comp:          string_30;
        when ident_2 =>
          enum_comp_2:          enumeration;
          string_comp_2:         string_30;
```

```
        when others =>
          char_comp_1,
          char_comp_2:          character;
        end case;
    end record;

end global_def;

    with global_def;
    use global_def;

package pack_1 is
-------------

    procedure proc_0;
    procedure proc_1(pointer_par_in:  in      record_pointer);
    procedure proc_2(int_par_in_out:  in out  one_to_fifty);
    procedure proc_3(pointer_par_out: out     record_pointer);

    int_glob:   integer;

end pack_1;

    with global_def;
    use global_def;

package pack_2 is
--------------

    procedure proc_6 (enum_par_in:      in      enumeration;
                      enum_par_out:     out     enumeration);

    procedure proc_7 (int_par_in_1,
                      int_par_in_2:     in      one_to_fifty;
                      int_par_out:      out     one_to_fifty);

    procedure proc_8 (array_par_in_out_1: in out array_1_dim_integer;
                      array_par_in_out_2: in out array_2_dim_integer;
                      int_par_in_1,
                      int_par_in_2:       in     integer);

    function func_1  (char_par_in_1,
                      char_par_in_2:      in     capital_letter)
                                                   return enumeration;

    function func_2  (string_par_in_1,
                      string_par_in_2:    in    string_30)
                                                   return boolean;

end pack_2;

with global_def, pack_1;
use global_def;

procedure A000091 is -- Dhrystone
--------------

begin
    pack_1.proc_0;        -- proc_0 is actually the main program, but it is
                          -- part of a package, and a program within a
                          -- package can not be designated as the main
                          -- program for execution. Therefore proc_0 is
                          -- activated by a call from "main".

end A000091 ;

with global_def,pack_2;
use global_def;
with cpu_time_clock;
with text_io;
with duration_io;

package body pack_1 is
```

```
-----------

    bool_glob:          boolean;
    char_glob_1,
    char_glob_2:        character;
    array_glob_1:       array_1_dim_integer;
    array_glob_2:       array_2_dim_integer;
    pointer_glob,
    pointer_glob_next:  record_pointer;

    start_time : duration ;
    stop_time : duration ;
    iteration_count : constant := 10_000 ;

    procedure proc_4;
    procedure proc_5;

procedure proc_0
is
    int_loc_1,
    int_loc_2,
    int_loc_3:          one_to_fifty;
    char_loc:           character;
    enum_loc:           enumeration;
    string_loc_1,
    string_loc_2:       string_30;

    begin
        -- initializations
        pack_1.pointer_glob_next := new record_type;

        pack_1.pointer_glob := new record_type
                            '(
                            pointer_comp => pack_1.pointer_glob_next,
                            discr        => ident_1,
                            enum_comp    => ident_3,
                            int_comp     => 40,
                            string_comp  => "DHRYSTONE PROGRAM, SOME STRING"
                               );

        string_loc_1 := "DHRYSTONE PROGRAM, 1'ST STRING";

---------------
-- start timer here
---------------
     start_time := cpu_time_clock ;
     for i in 1 .. iteration_count loop

        proc_5;
        proc_4;
         -- char_glob_1 = 'A', char_glob_2 = 'B', bool_glob = false

        int_loc_1 := 2;
        int_loc_2 := 3;
        string_loc_2 := "DHRYSTONE PROGRAM, 2'ND STRING";
        enum_loc := ident_2 ;
        bool_glob := not pack_2.func_2( string_loc_1,string_loc_2);
        -- bool_glob = true
        while int_loc_1 < int_loc_2 loop --loop body executed once
                    pragma TA_LOOP_BOUNDS(1,1);
           int_loc_3 := 5 * int_loc_1 - int_loc_2;
           -- int_loc_3 = 7
           pack_2.proc_7(int_loc_1,int_loc_2,int_loc_3);
           -- int_loc_3 = 7
           int_loc_1 := int_loc_1 + 1;
        end loop;
           -- int_loc_1 = 3
        pack_2.proc_8(array_glob_1,array_glob_2,int_loc_1,int_loc_3);
          -- int_glob = 5
        proc_1(pointer_glob);
        for char_index in 'A'..Char_glob_2 loop --loop body executed twice
            if enum_loc = pack_2.func_1(char_index,'C')
            then   -- not executed
```

```
                pack_2.proc_6(ident_1,enum_loc);
            end if;
        end loop;
          -- enum_loc = ident_1
          -- int_loc = 3, int_loc_2 = 3, int_loc_3 = 7
        int_loc_3 := int_loc_2 * int_loc_1;
        int_loc_2 := int_loc_3 / int_loc_1;
        int_loc_2 := 7 * ( int_loc_3 - int_loc_2 ) - int_loc_1;
        proc_2(int_loc_1);

     end loop ;
     stop_time := cpu_time_clock ;
     text_io.new_line;
     text_io.new_line;
     text_io.put_line("Test Name:   A000091            Class Name:  Composite");
     text_io.put("      ");
     duration_io.put((stop_time-start_time)*1000/iteration_count);
     text_io.put_line(" is time in milliseconds for one Dhrystone");
     text_io.put_line("Test Description:");
     text_io.put_line(" Reinhold P. Weicker's DHRYSTONE composite benchmark");
     text_io.new_line;
------------------
-- stop timer here
------------------

end proc_0;

procedure proc_1(pointer_par_in: in record_pointer) is -- executed once

        next_record: record_type
            renames pointer_par_in.pointer_comp.all; -- pointer_glob_next.all
begin
        next_record :=pointer_glob.all;
        pointer_par_in.int_comp := 5;
        next_record.int_comp := pointer_par_in.int_comp;
        next_record.pointer_comp:= pointer_par_in.pointer_comp;
        proc_3(next_record.pointer_comp);
           -- next_record.pointer_glob.pointer_comp = pointer_comp.next
        if next_record.discr = ident_1
        then  -- executed
            next_record.int_comp := 6;
            pack_2.proc_6(pointer_par_in.enum_comp,next_record.enum_comp);
            next_record.pointer_comp := pointer_glob.pointer_comp;
            pack_2.proc_7(next_record.int_comp,10,next_record.int_comp);
        else
            pointer_par_in.all := next_record;
        end if;
end proc_1;

procedure proc_2 ( int_par_in_out: in out one_to_fifty)
is -- executed once
   -- in_par_in_out = 3 becomes 7
    int_loc : one_to_fifty;
    enum_loc : enumeration;
begin
    int_loc := int_par_in_out + 10;
    loop
                 pragma TA_LOOP_BOUNDS(1,2);
        if char_glob_1 = 'A'
        then
            int_loc := int_loc - 1;
            int_par_in_out := int_loc - int_glob;
            enum_loc := ident_1;  -- true
        end if;
    exit when enum_loc = ident_1; -- true
    end loop;
end proc_2;

procedure proc_3(pointer_par_out: out record_pointer)
is -- executed once
        -- pointer_par_out becomes pointer_glob
begin
    if pointer_glob /= null
```

```
    then   -- executed
        pointer_par_out := pointer_glob.pointer_comp;
    else
        int_glob := 100;
    end if;
    pack_2.proc_7(10,int_glob,pointer_glob.int_comp);
end proc_3;

procedure proc_4
is
    bool_loc : boolean;
begin
    bool_loc := char_glob_1 = 'A';
    bool_loc := bool_loc or bool_glob;
    char_glob_2 := 'B';
end proc_4;

procedure proc_5
is
begin
    char_glob_1 := 'A';
    bool_glob := false;
end proc_5;

end pack_1;

    with global_def,pack_1; use global_def;
package body pack_2 is

function func_3(enum_par_in: in enumeration) return boolean;
        -- forward declaration
procedure proc_6(enum_par_in:  in  enumeration;
                 enum_par_out: out enumeration) is
begin
    enum_par_out := enum_par_in;
    if not func_3(enum_par_in) then
        enum_par_out := ident_4;
    end if;
    case enum_par_in is
        when ident_1 =>enum_par_out := ident_1;
        when ident_2 =>if pack_1.int_glob>100
                        then enum_par_out := ident_1;
                        else enum_par_out := ident_4;
                        end if;
        when ident_3 =>enum_par_out := ident_2; -- executed
        when ident_4 =>null;
        when ident_5 =>enum_par_out := ident_3;
    end case;
end proc_6;

procedure proc_7(int_par_in_1,
                int_par_in_2: in  one_to_fifty;
                int_par_out:  out one_to_fifty) is

int_loc : one_to_fifty;
begin
   int_loc := int_par_in_1 + 2;
   int_par_out := int_par_in_2 + int_loc;
end proc_7;

procedure proc_8 (array_par_in_out_1: in out array_1_dim_integer;
                  array_par_in_out_2: in out array_2_dim_integer;
                  int_par_in_1,
                  int_par_in_2:       in integer)
is

int_loc: one_to_fifty;
begin
    int_loc := int_par_in_1 + 5;
    array_par_in_out_1(int_loc) := int_par_in_2;
    array_par_in_out_1(int_loc + 1) :=
                                    array_par_in_out_1(int_loc);
    array_par_in_out_1(int_loc + 30) := int_loc;
```

```
    for int_index in int_loc..int_loc + 1 loop -- loop body executed twice
                pragma TA_LOOP_BOUNDS(2,2);
        array_par_in_out_2(int_loc,int_index) := int_loc ;
    end loop;
    array_par_in_out_2(int_loc,int_loc-1) :=
                            array_par_in_out_2(int_loc,int_loc-1) + 1;
    array_par_in_out_2(int_loc + 20,int_loc) :=
                            array_par_in_out_1(int_loc);
    pack_1.int_glob := 5;

end proc_8;

function func_1 (char_par_in_1,
                char_par_in_2: in capital_letter) return enumeration
is

char_loc_1, char_loc_2 : capital_letter;

begin
    char_loc_1 := char_par_in_1;
    char_loc_2 := char_loc_1;
    if char_loc_2 /= char_par_in_2 then
        return ident_1;
    else
        return ident_2;
    end if;
end func_1;

function func_2(string_par_in_1,
                string_par_in_2: in string_30) return boolean
is

int_loc: one_to_thirty;
char_loc: capital_letter;

begin
    int_loc := 2;
    while int_loc <= 2 loop
                pragma TA_LOOP_BOUNDS(1,1);
        if func_1(string_par_in_1(int_loc),
                string_par_in 2(int_loc+1)) = ident_1 then
            char_loc := 'A';
            int_loc := int_loc + 1;
        end if;
    end loop;
    if char_loc >='W' and char_loc < 'Z' then
        int_loc := 7;
    end if;
    if char_loc = 'X' then
        return true;
    else
        if string_par_in_1 > string_par_in_2 then
            int_loc := int_loc + 7;
            return true;
        else
            return false;
        end if;
    end if;
end func_2;

function func_3(enum_par_in: in enumeration) return boolean
is

    enum_loc: enumeration;
begin
    enum_loc := enum par_in;
    if enum_loc = ident_3 then
        return true;
    end if;
end func_3;

end pack_2;
-- Ada version of Whetstone Benchmark Program
```

```
-- This must be edited to "with" the compiler suppliers math routines
--  SIN, COS, ATAN, SQRT, EXP and LOG
--  These results may be interesting to compare to Z000093 that uses
--  a physically included, all Ada set of math routines
```

```
---------------------------------------------------------------------
--                                                                 --
--      WHETADA.ADA     distributed as A000092.ADA                 --
--                                                                 --
--    Ada version of the Whetstone Benchmark Program.              --
--    Reference: "Computer Journal" February 1976, pages 43-49     --
--               for description of benchmark and ALGOL60 version. --
-- Note: Procedure POUT is omitted.                                --
--                                                                 --
-- From Timing Studies using a synthetic Whetstone Benchmark       --
--      by Sam Harbaugh and John A. Forakis                        --
--                                                                 --
---------------------------------------------------------------------
--                                                                 --
-- Authors Disclaimer                                              --
-- "   The Whetstone measure deals only with the most basic scientific/ --
--  computational aspects of the languages and computers and no general --
--  conclusions should be drawn from this work. Application specific   --
--  benchmarks should be written and run by anyone needing to draw     --
--  conclusions reguarding suitability of languages, compilers and     --
--  hardware. This data is reported to stimulate interest and work in  --
--  run time benchmarking and in no way is meant to influence anyone's --
--  choice of languages or software in any situation "                 --
--                                                                 --
---------------------------------------------------------------------

with CPU_TIME_CLOCK ;
with TEXT_IO;   use TEXT_IO;
-- Change the following line to use the compiler vendors or manufacturers
-- math library.
with MATH_LIB; use MATH_LIB; -- manufacturers routines ( Meridian for Mac )

procedure A000092A is
 --pragma SUPPRESS(ACCESS_CHECK);         DO NOT USE PRAGMA SUPPRESS for PIWG
 --pragma SUPPRESS(DISCRIMINANT_CHECK);
 --pragma SUPPRESS(INDEX_CHECK);
 --pragma SUPPRESS(LENGTH_CHECK);
 --pragma SUPPRESS(RANGE_CHECK);
 --pragma SUPPRESS(DIVISION_CHECK);
 --pragma SUPPRESS(OVERFLOW_CHECK);
 --pragma SUPPRESS(STORAGE_CHECK);
 --pragma SUPPRESS(ELABORATION_CHECK);

  package REAL_IO is new FLOAT_IO(FLOAT);  use REAL_IO;

         subtype CYCLES is INTEGER range 10..50;

     procedure WHETSTONE(NO_OF_CYCLES : in CYCLES;
                         START_TIME,STOP_TIME: out FLOAT) is

     -- Calling procedure provides
     -- the encompassing loop count, NO_OF_CYCLES.

        type VECTOR is array (INTEGER range <>) of FLOAT;
        X1,X2,X3,X4,X,Y,Z : FLOAT;
        E1 : VECTOR(1..4);
        J,K,L : INTEGER;
        -- Set constants
        T  : constant := 0.499975;
        T1 : constant := 0.50025;
        T2 : constant := 2.0;
        -- Compute the execution frequency for the benchmark modules
        N1 : constant := 0;       --Module 1 not executed
        N2 : constant := 120;
        N3 : constant := 140;
        N4 : constant := 3450;
        N5 : constant := 0;      -- Module 5 not executed
        N6 : constant := 2100;
        N7 : constant := 320;
        N8 : constant := 8990;
        N9 : constant := 6160;
        N10: constant := 0;       -- Module 10 not executed
        N11: constant := 930;
```

```
procedure PA(E: in out VECTOR) is
-- tests computations with an array as a parameter
   J : INTEGER;
   -- T,T2 : FLOAT are global variables
   begin
     J:=0;
     loop
                         pragma TA_LOOP_BOUNDS(6,6);
                         E(1) := (E(1) + E(2) + E(3) - E(4)) * T;
                         E(2) := (E(1) + E(2) - E(3) + E(4)) * T;
                         E(3) := (E(1) - E(2) + E(3) + E(4)) * T;
                         E(4) := (-E(1) + E(2) + E(3) + E(4)) / T2;
                         J := J + 1;
                         exit when j >= 6;
                 end loop;
end PA;


procedure P0 is
-- tests computations with no parameters
-- T1,T2 : FLOAT are global
-- E1 : VECTOR(1..4) is global
-- J,K,L : INTEGER are global
   begin
     E1(J) := E1(K);
     E1(K) := E1(L);
     E1(L) := E1(J);
   end P0;


procedure P3(X,Y: in out FLOAT;  Z : out FLOAT) is
-- tests computations with simple identifiers as parameters
-- T,T2 : FLOAT are global
   begin
     X := T * (X + Y);
     Y := T * (X + Y);
     Z := (X + Y) / T2;
   end P3;


begin

  START_TIME := FLOAT(CPU_TIME_CLOCK);  --Get Whetstone start time

  CYCLE_LOOP:
  for CYCLE_NO in 1..NO_OF_CYCLES loop
     -- Module 1 : computations with simple identifiers
        X1 := 1.0;
        X2 := -1.0;
        X3 := -1.0;
        X4 := -1.0;
        for I in 1..N1 loop
           X1 := (X1 + X2 + X3 - X4) * T;
           X2 := (X1 + X2 - X3 + X4) * T;
           X3 := (X1 + X2 + X3 + X4) * T;
           X4 := (-X1 + X2 + X3 + X4) * T;
        end loop;
     -- end Module 1

     -- Module 2: computations with array elements
        E1(1) := 1.0;
        E1(2) := -1.0;
        E1(3) := -1.0;
        E1(4) := -1.0;
        for I in 1..N2 loop
           E1(1) := (E1(1) + E1(2) + E1(3) - E1(4)) * T;
           E1(2) := (E1(1) + E1(2) - E1(3) + E1(4)) * T;
           E1(3) := (E1(1) - E1(2) + E1(3) + E1(4)) * T;
           E1(4) := (-E1(1) + E1(2) + E1(3) + E1(4)) * T;
        end loop;
     -- end Module 2
```

```
-- Module 3 : passing an array as a parmeter
   for I in 1..N3 loop
        PA(E1);
   end loop;
-- end Module 3

-- Module 4 : performing conditional jumps
   J := 1;
   for I in 1..N4 loop
      if J=1 then
          J := 2;
      else
          J := 3;
      end if;
      if J>2 then
          J := 0;
      else
          J := 1;
      end if;
      if J<1 then
          J := 1;
      else
          J := 0;
      end if;
   end loop;
--end Module 4

-- Module 5 : omitted

-- Module 6 : performing integer arithmetic
   J := 1;
   K := 2;
   L := 3;
   for I in 1..N6 loop
      J := J * (K-J) * (L-K);
      K := L*K - (L-J) * K;
      L := (L-K) * (K+J);
      E1(L-1) := FLOAT(J+K+L);
      E1(K-1) := FLOAT(J*K*L);
   end loop;
-- end Module 6

-- Module 7 : performing computations using trigonometric
--            functions
   X := 0.5;
   Y := 0.5;
   for I in 1..N7 loop
     X := T*ATAN(T2*SIN(X)*COS(X)/(COS(X+Y)+COS(X-Y)-1.0));
     Y := T*ATAN(T2*SIN(Y)*COS(Y)/(COS(X+Y)+COS(X-Y)-1.0));
   end loop;
-- end Module 7

-- Module 8 : procedure calls with simple identifiers as
--            parameters
   X := 1.0;
   Y := 1.0;
   Z := 1.0;
   for I in 1..N8 loop
      P3(X,Y,Z);
   end loop;
-- end Module 8

-- Module 9 : array reference and procedure calls with no
--            parameters
   J := 1;
   K := 2;
   L := 3;
   E1(1) := 1.0;
   E1(2) := 2.0;
   E1(3) := 3.0;
   for I in 1..N9 loop
      P0;
   end loop;
```

```
      -- end Module 9

      -- Module 10 : integer arithmetic
         J := 2;
         K := 3;
         for I in 1..N10 loop
            J := J + K;
            K := K + J;
            J := K - J;
            K := K - J - J;
         end loop;
      -- end Module 10

      -- Module 11 : performing computations using standard
      --            mathematical functions
         X := 0.75;
         for I in 1..N11 loop
            X := SQRT(EXP(LN(X)/T1));
         end loop;
      -- end Moudle 11

    end loop CYCLE_LOOP;

    STOP_TIME := FLOAT(CPU_TIME_CLOCK);    --Get Whetstone stop time
end WHETSTONE;

procedure COMPUTE_WHETSTONE_KIPS is
   -- Variables used to control execution of benchmark and to
   -- compute the Whetstone rating :

    NO_OF_RUNS : constant := 5;   -- Number of times the benchmark is executed
    NO_OF_CYCLES : INTEGER; -- Number of times the group of benchmark
                            -- modules is executed
    -- I : INTEGER;
                -- Embedded (as 10) in "N" constants at beginning of WHETSTONE proc
      -- Factor weighting number of times each module loops
      -- A value of ten gives a total weight for modules of
      -- approximately one million Whetstone instructions
    START_TIME : FLOAT;
               -- Time at which execution of benchmark modules begins
    STOP_TIME : FLOAT;
              -- Time at which execution of benchmark modules ends
              -- (time for NO_OF_CYCLES)
    ELAPSED_TIME : FLOAT;
                 -- Time between START_TIME and STOP_TIME
    MEAN_TIME : FLOAT;     -- Average time per cycle
    RATING : FLOAT;    -- Thousands of Whetstone instructions per sec
    MEAN_RATING : FLOAT;     -- Average Whetstone rating
    INT_RATING : INTEGER;    -- Integer value of KWIPS

    begin
      NEW_LINE;
      PUT_LINE
      ("Test Name:   A000092                     Class Name: composite");

      MEAN_TIME := 0.0;
      MEAN_RATING := 0.0;
      NO_OF_CYCLES := 10;

      RUN_LOOP:
      for RUN_NO in 1..NO_OF_RUNS loop
         -- Call the Whetstone benchmark parocedure
         WHETSTONE(NO_OF_CYCLES,START_TIME,STOP_TIME);

         -- Compute and write elapsed time
         ELAPSED_TIME := STOP_TIME - START_TIME;

         -- Sum time in milliseconds per cycle
         MEAN_TIME := MEAN_TIME + (ELAPSED_TIME*1000.0)/
                      FLOAT(NO_OF_CYCLES);

         -- Calculate the Whetstone rating based on the time for
         -- the number of cycles just executed and write
```

```
      RATING := (1000.0 * FLOAT(NO_OF_CYCLES))/ELAPSED_TIME;

      -- Sum Whetstone rating
      MEAN_RATING := MEAN_RATING + RATING;
      INT_RATING := INTEGER(RATING);

      -- Reset NO_OF_CYCLES for next run using ten cycles more
      NO_OF_CYCLES := NO_OF_CYCLES + 10;
   end loop RUN_LOOP;

   -- Compute average time in millieseconds per cycle and write
   MEAN_TIME := MEAN_TIME/FLOAT(NO_OF_RUNS);

   NEW_LINE;  PUT("Average time per cycle : ");
   PUT(MEAN_TIME,5,2,0);   PUT_LINE(" milliseconds");

   -- Calculate average Whetstone rating and write
   MEAN_RATING := MEAN_RATING/FLOAT(NO_OF_RUNS);
   INT_RATING := INTEGER(MEAN_RATING);

   NEW_LINE;  PUT("Average Whetstone rating : ");
   PUT_LINE(INTEGER'IMAGE(INT_RATING) & " KWIPS");
   NEW_LINE;
   NEW_LINE;

end COMPUTE_WHETSTONE_KIPS;

begin
   COMPUTE_WHETSTONE_KIPS;
end A000092A;
```

```
-- PERFORMANCE MEASUREMENT : task creation and termination time
--                           1 task no entry
--                           task type in package, no select

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ;
package CREATE_PACK_1 is
  task type T1 is
  end T1 ;
  procedure P1 ; -- will create task, run task, and terminate task
end CREATE_PACK_1 ;

with CREATE_PACK_1 ; use CREATE_PACK_1 ;
with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION ; -- obtain stable measurement
with PIWG_IO ; -- output results

procedure C000001 is  -- main procedure to execute

  CPU_TIME : DURATION ; -- CPU time for one feature execution
  WALL_TIME : DURATION ; -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ; -- true when measurement stable


begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 5.0 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
 - Test loop
--
-- establish task create and terminate time

    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        P1 ; -- this has task that has global increment and call inside
      end loop ;
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;
```

```
--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "C000001" , "Tasking" ,
                        CPU_TIME , WALL_TIME , ITERATION_COUNT ,
    " Task create and terminate measurement " ,
    " with one task, no entries, when task is in a procedure" ,
    " using a task type in a package, no select statement, no loop, " ) ;

end C000001 ;

package body CREATE_PACK_1 is
  task body T1 is
  begin
    GLOBAL := GLOBAL + A_ONE ;
    REMOTE ;
  end T1 ;

  procedure P1 is
    T : T1 ; -- this creates the task, runs task to completion and terminates
  begin
    null ;
  end P1 ;

end CREATE_PACK_1 ;
```

```
-- PERFORMANCE MEASUREMENT : task creation and termination time
--                           1 task no entry
--                           task defined and used in procedure, no select

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ;
package CREATE_PACK_2 is
  procedure P1 ; -- will create task, run task, and terminate task
end CREATE_PACK_2 ;

with CREATE_PACK_2 ; use CREATE_PACK_2 ;
with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION ; -- obtain stable measurement
with PIWG_IO ; -- output results

procedure C000002 is  -- main procedure to execute

  CPU_TIME : DURATION ; -- CPU time for one feature execution
  WALL_TIME : DURATION ; -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ; -- true when measurement stable

begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 0.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
-- Test loop
--

    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        P1 ; -- this has task that has global increment and call inside
      end loop ;
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;

--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "C000002" , "Tasking" ,
                        CPU_TIME , WALL_TIME , ITERATION_COUNT ,
    " Task create and terminate time measurement. " ,
    " with one task, no entries when task is in a procedure," ,
    " task defined and used in procedure, no select statement, no loop " ) ;

end C000002 ;
```

```
package body CREATE_PACK_2 is

  procedure P1 is
--          this creates the task, runs task to completion and terminates
--          execution time for task taken out by control loop
    task T1 is
    end T1 ;

    task body T1 is
    begin
      GLOBAL := GLOBAL + A_ONE ;
      REMOTE ;
    end T1 ;

  begin
    null ;
  end P1 ;

end CREATE_PACK_2 ;
```

```
-- PERFORMANCE MEASUREMENT : operations on boolean arrays
--                           arrays are NOT packed
--                           operations on components in loop

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION ;                         -- obtain stable measurement
with PIWG_IO ;                           -- output results

procedure H000004 is  -- main procedure to execute

  CPU_TIME : DURATION ;           -- CPU time for one feature execution
  WALL_TIME : DURATION ;          -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ;              -- true when measurement stable
--
-- Boolean array declarations
--
  type UNPACKED_BIT_ARRAY is array ( NATURAL range <> ) of BOOLEAN;

  BIT_VALUE_1 : BOOLEAN := GLOBAL > 0;
  BIT_VALUE_2 : BOOLEAN := GLOBAL rem 2 = 0;
  BIT_VALUE_3 : BOOLEAN := GLOBAL <= 1;

  subtype UNPACKED_16 is UNPACKED_BIT_ARRAY ( 0 .. 15 );

  UNPACKED_1 : UNPACKED_16 := UNPACKED_16'( 0|3|6|9|12|15 => BIT_VALUE_1,
                                            1|5|7|11|13 => BIT_VALUE_2,
                                            others => BIT_VALUE_3 );
  UNPACKED_2 : UNPACKED_16 := UNPACKED_16'( 0..3 => BIT_VALUE_1,
                                            4..12 => BIT_VALUE_2,
                                            others => BIT_VALUE_3 );

begin  -- procedure H000004

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 0.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time
--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;
--
-- Test loop
--
    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE;
        for I in UNPACKED_16'RANGE loop
          UNPACKED_1( I ) := UNPACKED_2( I ) xor not UNPACKED_1( I );
        end loop;
        for I in UNPACKED_16'RANGE loop
          UNPACKED_2( I ) := UNPACKED_1( I ) or UNPACKED_2( I );
        end loop;
        for I in UNPACKED_16'RANGE loop
          UNPACKED_1( I ) := not( UNPACKED_1( I ) and UNPACKED 2( I ) );
        end loop;
        REMOTE;
      end loop ;
```

```
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
--
-- Be sure UNPACKED_1 has been computed
--
    if UNPACKED_1( GLOBAL rem 16 ) then
      GLOBAL := A_ONE;
      REMOTE;
    end if;

    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;

  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;
--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "H000004" , "Chapter 13" ,
                        CPU_TIME , WALL_TIME , ITERATION_COUNT ,
    " Time to perform standard boolean operations on arrays of booleans." ,
    " For this test the arrays are NOT PACKED with the pragma 'PACK.'" ,
    " For this test the operations are performed on components in a loop." ) ;

end H000004 ;
```

```
-- PERFORMANCE MEASUREMENT : Minimum procedure call and return time
--                           procedure local
--                           no parameters

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ;
with ITERATION ;
with PIWG_IO ;

procedure P000001 is  -- main procedure to execute

  CPU_TIME : DURATION ;
  WALL_TIME : DURATION ;
  CHECK_TIMES : constant := 100 ;
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ;
  ITS_OK : BOOLEAN ;

--

  procedure PROC_0 is  -- may be inlined thus zero time
  begin
    GLOBAL := GLOBAL + A_ONE ;
    REMOTE ;
  end ;

begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 0.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
-- Test loop
--
    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        PROC_0 ;  -- this has control global increment and call inside
      end loop ;
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , ITS_OK ) ;
    exit when ITS_OK ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;

--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "P000001" , "Procedure" ,
                        CPU_TIME , WALL_TIME , ITERATION_COUNT ,
     " Procedure call and return time ( may be zero if automatic inlining ) " ,
     " procedure is local " ,
     " no parameters " ) ;
end P000001 ;
```

```
-- PERFORMANCE MEASUREMENT : procedure call and return time
--                           procedure in package
--                           ten discrete "in" parameters

package PROC_PACKAGE_10 is
  procedure PROC_0 ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10 : in INTEGER ) ;
end PROC_PACKAGE_10 ;

with PROC_PACKAGE_10 ; use PROC_PACKAGE_10 ;
with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION :    obtain stable measurement
with PIWG_IO ; -- output results

procedure P000010 is  -- main procedure to execute
  CPU_TIME : DURATION ; -- CPU time for one feature execution
  WALL_TIME : DURATION ; -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ; -- true when measurement stable
  A1 : INTEGER := A_ONE ;
  A2 : INTEGER := A1 + A_ONE ;
  A3 : INTEGER := A2 + A_ONE ;
  A4 : INTEGER := A3 + A_ONE ;
  A5 : INTEGER := A4 + A_ONE ;
  A6 : INTEGER := A5 + A_ONE ;
  A7 : INTEGER := A6 + A_ONE ;
  A8 : INTEGER := A7 + A_ONE ;
  A9 : INTEGER := A8 + A_ONE ;
  A10 : INTEGER := A9 + A_ONE ;

begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 0.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A1+A2+A3+A4+A5+A6+A7-A8-A9-A10 ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
-- Test loop
--
    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        PROC_0 ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10 ) ;
        -- this has control global increment and call inside
      end loop ;
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;
```

```
--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "P000010" , "Procedure" ,
                      CPU_TIME , WALL_TIME , ITERATION_COUNT ,
     " Procedure call and return time measurement" ,
     " Compare to P000005 " ,
     " 10 parameters, in INTEGER " ) ;

end P000010 ;


with REMOTE_GLOBAL ; use REMOTE_GLOBAL ;
package body PROC_PACKAGE_10 is -- compare to P000005
  procedure PROC_0 ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10 : in INTEGER ) is
  begin
    GLOBAL := GLOBAL + A1+A2+A3+A4+A5+A6+A7-A8-A9-A10 ;
    REMOTE ;
  end ;
end PROC_PACKAGE_10 ;
```

```
-- PERFORMANCE MEASUREMENT : Minimum entry call and return time
--                           task inside procedure
--                           1 task 1 entry
--                           no select, do..end

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION ; -- obtain stable measurement
with PIWG_IO ; -- output results

procedure T000001 is  -- main procedure to execute

  CPU_TIME : DURATION ; -- CPU time for one feature execution
  WALL_TIME : DURATION ; -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ; -- true when measurement stable

--
  task T1 is
    entry E1 ;
  end T1 ;

  task body T1 is
  begin
    loop
      accept E1 do
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end E1 ;
    end loop ;
  end ;

begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 00.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
-- Test loop
--
    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        T1.E1 ; -- this has control global increment and call inside
      end loop ;
    end loop ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;
```

```
--
-- Printout
--
   PIWG_IO.PIWG_OUTPUT ( "T000001" , "Tasking" ,
                         CPU_TIME , WALL_TIME , ITERATION_COUNT ,
     " Minimum rendezvous, entry call and return time " ,
     " 1 task 1 entry , task inside procedure " ,
     " no select " ) ;

  abort T1 ;

end T000001 ;
```

```
-- PERFORMANCE MEASUREMENT : tasks entry call and return time
--                          1 task 2 entries
--                          one select statement

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ;
package TASK_PACK_4 is
  task T1 is
    entry E1 ;
    entry E2 ;
  end T1 ;
end TASK_PACK_4 ;

with TASK_PACK_4 ; use TASK_PACK_4 ;
with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; -- control optimization
with ITERATION ; -- obtain stable measurement
with PIWG_IO ; -- output results

procedure T000004 is  -- main procedure to execute
  CPU_TIME : DURATION ; -- CPU time for one feature execution
  WALL_TIME : DURATION ; -- WALL time for one feature execution
  CHECK_TIMES : constant := 100 ; -- inside loop count and check
  ITERATION_COUNT : ITERATION.ITERATION_COUNTS ; -- set and varied by ITERATION package
  STABLE : BOOLEAN ; -- true when measurement stable
  CASE_COUNT : constant := 2 ;

begin

  ITERATION.START_CONTROL ;  -- dummy to bring in pages on some machines

  delay 0.5 ;  -- wait for stable enviornment on some machines

  ITERATION.INITIALIZE ( ITERATION_COUNT ) ;

  loop  -- until stable measurement, ITERATION_COUNT increases each time

--
-- Control loop
--
    ITERATION.START_CONTROL ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        GLOBAL := GLOBAL + A_ONE ;
        REMOTE ;
      end loop ;
    end loop ;
    ITERATION.STOP_CONTROL ( GLOBAL , CHECK_TIMES ) ;

--
-- Test loop
--
    ITERATION.START_TEST ;
    for J in 1 .. ITERATION_COUNT loop
      GLOBAL := 0 ;
      for INSIDE_LOOP in 1 .. CHECK_TIMES loop
        T1.E1 ; -- this has control global increment and call inside
        T1.E2 ; -- this has control global increment and call inside
      end loop ;
    end loop ;
    GLOBAL := GLOBAL / CASE_COUNT ;
    ITERATION.STOP_TEST ( GLOBAL , CHECK_TIMES ) ;
    ITERATION.TEST_STABLE ( ITERATION_COUNT , STABLE ) ;
    exit when STABLE ;
  end loop ;
--
  ITERATION.FEATURE_TIMES ( CPU_TIME , WALL_TIME ) ;
  CPU_TIME := DURATION ( CPU_TIME / CASE_COUNT ) ;
  WALL_TIME := DURATION ( WALL_TIME / CASE_COUNT ) ;
```

```
--
-- Printout
--
  PIWG_IO.PIWG_OUTPUT ( "T000004" , "Tasking" ,
                        CPU_TIME , WALL_TIME , ITERATION_COUNT ,
     " Task entry call and return time measured" ,
     " One tasks active, two entries, tasks in a package " ,
     " using select statement " ) ;

  abort T1 ;
end T000004 ;

package body TASK_PACK_4 is
  task body T1 is
  begin
    loop
      select
        accept E1 do
          GLOBAL := GLOBAL + A_ONE ;
          REMOTE ;
        end E1 ;
      or
        accept E2 do
          GLOBAL := GLOBAL + A_ONE ;
          REMOTE ;
        end E2 ;
      end select ;
    end loop ;
  end T1 ;

end TASK_PACK_4 ;
```

Appendix C

Selected DIANA Representations of Test Programs

The timed fragment from P000001 and the entirety of SimpleTasks is included here in DIANA form. This DIANA form is that used on the Rational machine. It represents as a bracketed list with a node type tag, non-structural attributes, and child nodes (structural attributes) in that order, e.g., [dn_type attr1 attr2 [child1] [child2]]. The hexadecimal numbers to the left are memory addresses for the nodes and can be *ignored*. Semantic attributes of the form sm_attr = [dn_tag ^] represent a pointer to a specific existing node of the type indicated.

P000001:

```
1FC910A_10C7B:                      [DN_LOOP
                                        lx_line_count = 7
1FC910A_10E82:                          [DN_FOR
1FC910A_10EFB:                              [DN_ITERATION_ID
                                                SM_SEQNUM = 1
                                                SM_PARENT = [DN_PROC_ID ^]
                                                lx_symrep = "J"
                                                sm_obj_type = [DN_RANGE ^]
                                            ]
1FC910A_1101B:                              [DN_RANGE
                                                sm_base_type = [DN_INTEGER ^]
1FC910A_110B7:                                  [DN_NUMERIC_LITERAL
                                                    lx_numrep = "1"
                                                    sm_exp_type = [DN_INTEGER ^]
                                                    sm_value = 1
                                                ]
1FC910A_11136:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "ITERATION_COUNT"
                                                    sm_defn = [DN_VAR_ID ^]
                                                    sm_exp_type = [DN_CONSTRAINED ^]
                                                    sm_value = No value
                                                ]
                                            ]
                                        ]
1FC910A_11FE1:                          [DN_STM_S
                                            lx_line_count = 5
```

```
1FC910A_111F3:      [DN_ASSIGN
                        lx_line_count = 1
1FC910A_1127D:          [DN_USED_OBJECT_ID
                            lx_symrep = "GLOBAL"
                            sm_defn = [DN_VAR_ID ^]
                            sm_exp_type = [DN_CONSTRAINED ^]
                            sm_value = Uninitialized
                        ]
1FC910A_1131F:          [DN_NUMERIC_LITERAL
                            lx_numrep = "0"
                            sm_exp_type = [DN_INTEGER ^]
                            sm_value = 0
                        ]
                    ]
1FC910A_113B9:      [DN_LOOP
                        lx_line_count = 4
1FC910A_115C0:          [DN_FOR
1FC910A_11639:              [DN_ITERATION_ID
                                SM_SEQNUM = 1
                                SM_PARENT = [DN_PROC_ID ^]
                                lx_symrep = "INSIDE_LOOP"
                                sm_obj_type = [DN_RANGE ^]
                            ]
1FC910A_11759:              [DN_RANGE
                                sm_base_type = [DN_INTEGER ^]
1FC910A_117F5:                  [DN_NUMERIC_LITERAL
                                    lx_numrep = "1"
                                    sm_exp_type = [DN_INTEGER ^]
                                    sm_value = 1
                                ]
1FC910A_11874:                  [DN_USED_OBJECT_ID
                                    lx_symrep = "CHECK_TIMES"
                                    sm_defn = [DN_NUMBER_ID ^]
                                    sm_exp_type = [DN_INTEGER ^]
                                    sm_value = 100
                                ]
                            ]
                        ]
1FC910A_11F71:          [DN_STM_S
                            lx_line_count = 2
1FC910A_11931:              [DN_ASSIGN
                                lx_line_count = 1
1FC910A_119BB:                  [DN_USED_OBJECT_ID
                                    lx_symrep = "GLOBAL"
                                    sm_defn = [DN_VAR_ID ^]
                                    sm_exp_type = [DN_CONSTRAINED ^]
                                    sm_value = Uninitialized
                                ]
1FC910A_11A5D:                  [DN_FUNCTION_CALL
                                    sm_exp_type = [DN_INTEGER ^]
                                    sm_value = Uninitialized
                                    sm_normalized_param_s = [DN_EXP_S ^]
                                    lx_prefix = FALSE
1FC910A_11B1C:                      [DN_USED_BLTN_OP
                                        SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                        lx_symrep = "+"
                                        sm_operator = INTEGER_ADD
                                    ]
1FC910A_11D7D:                      [DN_PARAM_ASSOC_S
1FC910A_11BCF:                          [DN_USED_OBJECT_ID
                                          lx_symrep = "GLOBAL"
                                          sm_defn = [DN_VAR_ID ^]
                                          sm_exp_type = [DN_CONSTRAINED ^]
                                          sm_value = Uninitialized
                                        ]
1FC910A_11CA6:                          [DN_USED_OBJECT_ID
                                          lx_symrep = "A_ONE"
                                          sm_defn = [DN_VAR_ID ^]
                                          sm_exp_type = [DN_CONSTRAINED ^]
                                          sm_value = Uninitialized
                                        ]
                                    ]
                                 ]
```

```
                                        ]
1FC910A_11DF7:                          [DN_PROCEDURE_CALL
                                            lx_line_count = 1
                                            sm_normalized_param_s = [DN_EXP_S ^]
1FC910A_11E9B:                              [DN_USED_NAME_ID
                                                lx_symrep = "REMOTE"
                                                sm_defn = [DN_PROC_ID ^]
                                            ]
1FC910A_11F12:                              [DN_PARAM_ASSOC_S]
                                        ]
                                    ]
                                ]
                            ]
                        ]
```

SimpleTasks:

```
1F77D0A_AD12:   [DN_COMP_UNIT
                    lx_line_count = 55
                    SM_ID_TABLE =
1F77D0A_CE56:       [DN_CONTEXT
                        lx_line_count = 0
                    ]
1F77D0A_CEC6:       [DN_SUBPROGRAM_BODY
                        SM_FORWARD = [DN_SUBPROGRAM_DECL ^]
                        lx_line_count = 55
1F77D0A_CF6A:           [DN_PROC_ID
                            DD_TRUST_ME = 2
                            lx_symrep = "DOIT"
                            sm_spec = [DN_PROCEDURE ^]
                            sm_body = [DN_BLOCK ^]
                            sm_stub = null
                            sm_first = [DN_PROC_ID ^]
                        ]
1F77D0A_D0AC:           [DN_PROCEDURE
1F77D0A_D10B:               [DN_PARAM_S
                                lx_line_count = 0
                                SM_ID_TABLE =
                            ]
                        ]
1F77D0A_D19B:           [DN_BLOCK
                            POST_COMMENT_HEIGHT = -2
                            PRE_COMMENT_HEIGHT = -2
                            lx_line_count = 53
1F77D0A_13D74:              [DN_ITEM_S
                                lx_line_count = 49
1F77D0A_D25A:                   [DN_VAR
                                    lx_line_count = 1
1F77D0A_D554:                       [DN_ID_S
1F77D0A_D2FE:                           [DN_VAR_ID
                                            SM_PARENT = [DN_PROC_ID ^]
                                            lx_symrep = "COUNT_A"
                                            sm_obj_type = [DN_CONSTRAINED ^]
                                            sm_obj_def = [DN_NUMERIC_LITERAL ^]
                                        ]
1F77D0A_D429:                           [DN_VAR_ID
                                            SM_PARENT = [DN_PROC_ID ^]
                                            lx_symrep = "COUNT_B"
                                            sm_obj_type = [DN_CONSTRAINED ^]
                                            sm_obj_def = [DN_NUMERIC_LITERAL ^]
                                        ]
                                    ]
1F77D0A_D5B3:                       [DN_CONSTRAINED
                                        sm_type_struct = [DN_INTEGER ^]
                                        sm_base_type = [DN_INTEGER ^]
                                        sm_constraint = [DN_RANGE ^]
1F77D0A_D695:                           [DN_USED_NAME_ID
                                            lx_symrep = "INTEGER"
                                            sm_defn = [DN_TYPE_ID ^]
                                        ]
                                        [DN_VOID]
                                    ]
1F77D0A_D70C:                       [DN_NUMERIC_LITERAL
                                        lx_numrep = "0"
                                        sm_exp_type = [DN_INTEGER ^]
                                        sm_value = 0
                                    ]
                                ]
1F77D0A_D7A6:                   [DN_VAR
                                    POST_COMMENT_HEIGHT = -2
                                    lx_line_count = 2
1F77D0A_D975:                       [DN_ID_S
1F77D0A_D84A:                           [DN_VAR_ID
                                            SM_PARENT = [DN_PROC_ID ^]
                                            lx_symrep = "RESULT_C"
                                            sm_obj_type = [DN_CONSTRAINED ^]
                                            sm_obj_def = [DN_NUMERIC_LITERAL ^]
                                        ]
                                    ]
1F77D0A_D9D4:                       [DN_CONSTRAINED
                                        sm_type_struct = [DN_INTEGER ^]
                                        sm_base_type = [DN_INTEGER ^]
                                        sm_constraint = [DN_RANGE ^]
1F77D0A_DAB6:                           [DN_USED_NAME_ID
                                            lx_symrep = "INTEGER"
                                            sm_defn = [DN_TYPE_ID ^]
                                        ]
                                        [DN_VOID]
                                    ]
1F77D0A_DB2D:                       [DN_NUMERIC_LITERAL
                                        lx_numrep = "6"
                                        sm_exp_type = [DN_INTEGER ^]
                                        sm_value = 6
                                    ]
                                ]
1F77D0A_DC0A:                   [DN_TASK_DECL
                                    POST_COMMENT_HEIGHT = -2
```

```
                                  lx_line_count = 3
1F77D0A_DC94:                     [DN_VAR_ID
                                      SM_PARENT = [DN_PROC_ID ^]
                                      lx_symrep = "TASK_A"
                                      sm_obj_type = [DN_TASK_SPEC ^]
                                      sm_obj_def = [DN_TASK_SPEC ^]
                                  ]
1F77D0A_DD8A:                     [DN_TASK_SPEC
                                      lx_line_count = 0
                                      sm_body = [DN_BLOCK ^]
1F77D0A_DE1D:                         [DN_DECL_S
                                          LX_VERBOSE = TRUE
                                          lx_line_count = 0
                                          SM_ID_TABLE =
                                      ]
                                  ]
                              ]
1F77D0A_DF3F:                 [DN_TASK_DECL
                                  POST_COMMENT_HEIGHT = -2
                                  lx_line_count = 3
1F77D0A_DFC9:                     [DN_VAR_ID
                                      SM_PARENT = [DN_PROC_ID ^]
                                      lx_symrep = "TASK_B"
                                      sm_obj_type = [DN_TASK_SPEC ^]
                                      sm_obj_def = [DN_TASK_SPEC ^]
                                  ]
1F77D0A_E0BF:                     [DN_TASK_SPEC
                                      lx_line_count = 0
                                      sm_body = [DN_BLOCK ^]
1F77D0A_E152:                         [DN_DECL_S
                                          LX_VERBOSE = TRUE
                                          lx_line_count = 0
                                          SM_ID_TABLE =
                                      ]
                                  ]
                              ]
1F77D0A_E274:                 [DN_TASK_DECL
                                  POST_COMMENT_HEIGHT = -2
                                  lx_line_count = 6
1F77D0A_E2FE:                     [DN_VAR_ID
                                      SM_PARENT = [DN_PROC_ID ^]
                                      lx_symrep = "TASK_C"
                                      sm_obj_type = [DN_TASK_SPEC ^]
                                      sm_obj_def = [DN_TASK_SPEC ^]
                                  ]
1F77D0A_E3F4:                     [DN_TASK_SPEC
                                      lx_line_count = 0
                                      sm_body = [DN_BLOCK ^]
1F77D0A_EDC0:                         [DN_DECL_S
                                          lx_line_count = 3
1F77D0A_E487:                             [DN_SUBPROGRAM_DECL
                                              lx_line_count = 1
1F77D0A_E52B:                                 [DN_ENTRY_ID
                                                  SM_SEQNUM = 1
                                                  SM_PARENT = [DN_VAR_ID ^]
                                                  lx_symrep = "ENTRY_A"
                                                  sm_spec = [DN_ENTRY ^]
                                              ]
1F77D0A_E64B:                                 [DN_ENTRY
                                                  lx_line_count = 0
                                                  [DN_VOID]
1F77D0A_E6D5:                                     [DN_PARAM_S
                                                      lx_line_count = 0
                                                      SM_ID_TABLE =
                                                  ]
                                              ]
                                              [DN_VOID]
                                          ]
1F77D0A_E79A:                             [DN_SUBPROGRAM_DECL
                                              lx_line_count = 1
1F77D0A_E83E:                                 [DN_ENTRY_ID
                                                  SM_SEQNUM = 2
                                                  SM_PARENT = [DN_VAR_ID ^]
                                                  lx_symrep = "ENTRY_B"
                                                  sm_spec = [DN_ENTRY ^]
                                              ]
1F77D0A_E95E:                                 [DN_ENTRY
                                                  lx_line_count = 0
                                                  [DN_VOID]
1F77D0A_E9E8:                                     [DN_PARAM_S
                                                      lx_line_count = 0
                                                      SM_ID_TABLE =
                                                  ]
                                              ]
                                              [DN_VOID]
                                          ]
1F77D0A_EAAD:                             [DN_SUBPROGRAM_DECL
                                              lx_line_count = 1
1F77D0A_EB51:                                 [DN_ENTRY_ID
                                                  SM_SEQNUM = 3
                                                  SM_PARENT = [DN_VAR_ID ^]
                                                  lx_symrep = "DONE"
                                                  sm_spec = [DN_ENTRY ^]
                                              ]
```

```
1F77D0A_EC71:                    [DN_ENTRY
                                     lx_line_count = 0
                                     [DN_VOID]
1F77D0A_FCFB:                        [DN_PARAM_S
                                         lx_line_count = 0
                                         SM_ID_TABLE =
                                     ]
                                 ]
                                 [DN_VOID]
                             ]
                         ]
                     ]
                 ]
1F77D0A_F14A:    [DN_TASK_BODY
                     SM_FORWARD = [DN_TASK_DECL ^]
                     lx_line_count = 9
1F77D0A_F1D4:        [DN_TASK_BODY_ID
                         lx_symrep = "TASK_A"
                         sm_spec = [DN_TASK_SPEC ^]
                         sm_body = [DN_BLOCK ^]
                         sm_stub = null
                         sm_first = [DN_VAR_ID ^]
                     ]
1F77D0A_F2C3:        [DN_BLOCK
                         POST_COMMENT_HEIGHT = -2
                         lx_line_count = 8
1F77D0A_F367:            [DN_ITEM_S
                             lx_line_count = 0
                             SM_ID_TABLE =
                         ]
1F77D0A_106E3:           [DN_STM_S
                             lx_line_count = 6
1F77D0A_F412:                [DN_LOOP
                                 lx_line_count = 3
1F77D0A_F619:                    [DN_FOR
1F77D0A_F692:                        [DN_ITERATION_ID
                                         SM_SEQNUM = 1
                                         SM_PARENT = [DN_VAR_ID ^]
                                         lx_symrep = "I"
                                         sm_obj_type = [DN_RANGE ^]
                                     ]
1F77D0A_F7B2:                        [DN_RANGE
                                         sm_base_type = [DN_INTEGER ^]
1F77D0A_F84E:                            [DN_NUMERIC_LITERAL
                                             lx_numrep = "1"
                                             sm_exp_type = [DN_INTEGER ^]
                                             sm_value = 1
                                         ]
1F77D0A_F8CD:                            [DN_NUMERIC_LITERAL
                                             SM_INT_VALUE = 1000
                                             lx_numrep = "1000"
                                             sm_exp_type = [DN_INTEGER ^]
                                             sm_value = 1000
                                         ]
                                     ]
                                 ]
1F77D0A_FE62:                    [DN_STM_S
                                     lx_line_count = 1
1F77D0A_F9DA:                        [DN_ASSIGN
                                         lx_line_count = 1
1F77D0A_FA64:                            [DN_USED_OBJECT_ID
                                             lx_symrep = "COUNT_A"
                                             sm_defn = [DN_VAR_ID ^]
                                             sm_exp_type = [DN_CONSTRAINED ^]
                                             sm_value = Uninitialized
                                         ]
1F77D0A_FB06:                            [DN_FUNCTION_CALL
                                             sm_exp_type = [DN_INTEGER ^]
                                             sm_value = Uninitialized
                                             sm_normalized_param_s = [DN_EXP_S ^]
                                             lx_prefix = FALSE
1F77D0A_FBC5:                                [DN_USED_BLTN_OP
                                                 SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                                 lx_symrep = "+"
                                                 sm_operator = INTEGER_ADD
                                             ]
1F77D0A_FC03:                                [DN_PARAM_ASSOC_S
1F77D0A_FC78:                                    [DN_USED_OBJECT_ID
                                                     lx_symrep = "COUNT_A"
                                                     sm_defn = [DN_VAR_ID ^]
                                                     sm_exp_type = [DN_CONSTRAINED ^]
                                                     sm_value = Uninitialized
                                                 ]
1F77D0A_FD4F:                                    [DN_NUMERIC_LITERAL
                                                     lx_numrep = "1"
                                                     sm_exp_type = [DN_INTEGER ^]
                                                     sm_value = 1
                                                 ]
                                             ]
                                         ]
                                     ]
                                 ]
                             ]
1F77D0A_FED2:                [DN_ENTRY_CALL
```

```
                                        SM_ORIGINAL_NODE = [DN_PROCEDURE_CALL ^]
                                        lx_line_count = 1
                                        sm_normalized_param_s = [DN_EXP_S ^]
1F77D0A_FF76:                           [DN_SELECTED
                                            sm_exp_type = null
                                            sm_value = Uninitialized
1F77D0A_1001A:                              [DN_USED_OBJECT_ID
                                                lx_symrep = "TASK_C"
                                                sm_defn = [DN_VAR_ID ^]
                                                sm_exp_type = [DN_TASK_SPEC ^]
                                                sm_value = Uninitialized
                                            ]
1F77D0A_100BC:                              [DN_USED_NAME_ID
                                                lx_symrep = "ENTRY_A"
                                                sm_defn = [DN_ENTRY_ID ^]
                                            ]
                                        ]
1F77D0A_10133:                          [DN_PARAM_ASSOC_S]
                                    ]
1F77D0A_10238:                      [DN_ASSIGN
                                        lx_line_count = 1
1F77D0A_102C2:                          [DN_USED_OBJECT_ID
                                            lx_symrep = "COUNT_A"
                                            sm_defn = [DN_VAR_ID ^]
                                            sm_exp_type = [DN_CONSTRAINED ^]
                                            sm_value = Uninitialized
                                        ]
1F77D0A_10364:                          [DN_FUNCTION_CALL
                                            sm_exp_type = [DN_INTEGER ^]
                                            sm_value = Uninitialized
                                            sm_normalized_param_s = [DN_EXP_S ^]
                                            lx_prefix = FALSE
1F77D0A_10423:                              [DN_USED_BLTN_OP
                                                SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                                lx_symrep = "+"
                                                sm_operator = INTEGER_ADD
                                            ]
1F77D0A_10684:                              [DN_PARAM_ASSOC_S
1F77D0A_104D6:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "COUNT_A"
                                                    sm_defn = [DN_VAR_ID ^]
                                                    sm_exp_type = [DN_CONSTRAINED ^]
                                                    sm_value = Uninitialized
                                                ]
1F77D0A_105AD:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "COUNT_A"
                                                    sm_defn = [DN_VAR_ID ^]
                                                    sm_exp_type = [DN_CONSTRAINED ^]
                                                    sm_value = Uninitialized
                                                ]
                                            ]
                                        ]
                                    ]
                                ]
1F77D0A_10753:                  [DN_ALTERNATIVE_S
                                    lx_line_count = 0
                                ]
                            ]
                        ]
1F77D0A_10891:          [DN_TASK_BODY
                            SM_FORWARD = [DN_TASK_DECL ^]
                            lx_line_count = 9
1F77D0A_1091B:              [DN_TASK_BODY_ID
                                lx_symrep = "TASK_B"
                                sm_spec = [DN_TASK_SPEC ^]
                                sm_body = [DN_BLOCK ^]
                                sm_stub = null
                                sm_first = [DN_VAR_ID ^]
                            ]
1F77D0A_10A0A:              [DN_BLOCK
                                POST_COMMENT_HEIGHT = -2
                                lx_line_count = 8
1F77D0A_10AAE:                  [DN_ITEM_S
                                    lx_line_count = 0
                                    SM_ID_TABLE =
                                ]
1F77D0A_11DDA:                  [DN_STM_S
                                    lx_line_count = 6
1F77D0A_10B59:                      [DN_LOOP
                                        lx_line_count = 3
1F77D0A_10D60:                          [DN_FOR
1F77D0A_10DD9:                              [DN_ITERATION_ID
                                                SM_SEQNUM = 1
                                                SM_PARENT = [DN_VAR_ID ^]
                                                lx_symrep = "I"
                                                sm_obj_type = [DN_RANGE ^]
                                            ]
1F77D0A_10EF9:                              [DN_RANGE
                                                sm_base_type = [DN_INTEGER ^]
1F77D0A_10F95:                                  [DN_NUMERIC_LITERAL
                                                    lx_numrep = "1"
                                                    sm_exp_type = [DN_INTEGER ^]
                                                    sm_value = 1
                                                ]
```

```
1F77D0A_11014:                          [DN_NUMERIC_LITERAL
                                            lx_numrep = "100"
                                            sm_exp_type = [DN_INTEGER ^]
                                            sm_value = 100
                                        ]
                                    ]
                                ]
1F77D0A_11559:                  [DN_STM_S
                                    lx_line_count = 1
1F77D0A_110AE:                      [DN_ASSIGN
                                        lx_line_count = 1
1F77D0A_11138:                          [DN_USED_OBJECT_ID
                                            lx_symrep = "COUNT_B"
                                            sm_defn = [DN_VAR_ID ^]
                                            sm_exp_type = [DN_CONSTRAINED ^]
                                            sm_value = Uninitialized
                                        ]
1F77D0A_111DA:                          [DN_FUNCTION_CALL
                                            sm_exp_type = [DN_INTEGER ^]
                                            sm_value = Uninitialized
                                            sm_normalized_param_s = [DN_EXP_S ^]
                                            lx_prefix = FALSE
1F77D0A_11299:                              [DN_USED_BLTN_OP
                                                SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                                lx_symrep = "+"
                                                sm_operator = INTEGER_ADD
                                            ]
1F77D0A_114FA:                              [DN_PARAM_ASSOC_S
1F77D0A_1134C:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "COUNT_B"
                                                    sm_defn = [DN_VAR_ID ^]
                                                    sm_exp_type = [DN_CONSTRAINED ^]
                                                    sm_value = Uninitialized
                                                ]
1F77D0A_11423:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "I"
                                                    sm_defn = [DN_ITERATION_ID ^]
                                                    sm_exp_type = [DN_RANGE ^]
                                                    sm_value = Uninitialized
                                                ]
                                            ]
                                        ]
                                    ]
                                ]
                            ]
1F77D0A_115C9:              [DN_ENTRY_CALL
                                SM_ORIGINAL_NODE = [DN_PROCEDURE_CALL ^]
                                lx_line_count = 1
                                sm_normalized_param_s = [DN_EXP_S ^]
1F77D0A_1166D:                  [DN_SELECTED
                                    sm_exp_type = null
                                    sm_value = Uninitialized
1F77D0A_11711:                      [DN_USED_OBJECT_ID
                                        lx_symrep = "TASK_C"
                                        sm_defn = [DN_VAR_ID ^]
                                        sm_exp_type = [DN_TASK_SPEC ^]
                                        sm_value = Uninitialized
                                    ]
1F77D0A_117B3:                      [DN_USED_NAME_ID
                                        lx_symrep = "ENTRY_B"
                                        sm_defn = [DN_ENTRY_ID ^]
                                    ]
                                ]
1F77D0A_1182A:                  [DN_PARAM_ASSOC_S]
                            ]
1F77D0A_1192F:              [DN_ASSIGN
                                lx_line_count = 1
1F77D0A_119B9:                  [DN_USED_OBJECT_ID
                                    lx_symrep = "COUNT_B"
                                    sm_defn = [DN_VAR_ID ^]
                                    sm_exp_type = [DN_CONSTRAINED ^]
                                    sm_value = Uninitialized
                                ]
1F77D0A_11A5B:                  [DN_FUNCTION_CALL
                                    sm_exp_type = [DN_INTEGER ^]
                                    sm_value = Uninitialized
                                    sm_normalized_param_s = [DN_EXP_S ^]
                                    lx_prefix = FALSE
1F77D0A_11B1A:                      [DN_USED_BLTN_OP
                                        SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                        lx_symrep = "+"
                                        sm_operator = INTEGER_ADD
                                    ]
1F77D0A_11D7B:                      [DN_PARAM_ASSOC_S
1F77D0A_11BCD:                          [DN_USED_OBJECT_ID
                                            lx_symrep = "COUNT_B"
                                            sm_defn = [DN_VAR_ID ^]
                                            sm_exp_type = [DN_CONSTRAINED ^]
                                            sm_value = Uninitialized
                                        ]
1F77D0A_11CA4:                          [DN_USED_OBJECT_ID
                                            lx_symrep = "COUNT_B"
                                            sm_defn = [DN_VAR_ID ^]
                                            sm_exp_type = [DN_CONSTRAINED ^]
```

```
                                                  sm_value = Uninitialized
                                              ]
                                          ]
                                      ]
                                  ]
                              ]
1F77D0A_11E4A:                [DN_ALTERNATIVE_S
                                  lx_line_count = 0
                              ]
                          ]
                      ]
1F77D0A_11F88:        [DN_TASK_BODY
                          SM_FORWARD = [DN_TASK_DECL ^]
                          lx_line_count = 16
1F77D0A_12012:            [DN_TASK_BODY_ID
                              lx_symrep = "TASK_C"
                              sm_spec = [DN_TASK_SPEC ^]
                              sm_body = [DN_BLOCK ^]
                              sm_stub = null
                              sm_first = [DN_VAR_ID ^]
                          ]
1F77D0A_12101:            [DN_BLOCK
                              POST_COMMENT_HEIGHT = -2
                              lx_line_count = 15
1F77D0A_121A5:                [DN_ITEM_S
                                  lx_line_count = 0
                                  SM_ID_TABLE =
                              ]
1F77D0A_13BC6:                [DN_STM_S
                                  lx_line_count = 13
1F77D0A_12250:                    [DN_LOOP
                                      lx_line_count = 11
1F77D0A_12457:                        [DN_FOR
1F77D0A_124D0:                            [DN_ITERATION_ID
                                              SM_SEQNUM = 1
                                              SM_PARENT = [DN_VAR_ID ^]
                                              lx_symrep = "I"
                                              sm_obj_type = [DN_RANGE ^]
                                          ]
1F77D0A_125F0:                            [DN_RANGE
                                              sm_base_type = [DN_INTEGER ^]
1F77D0A_1268C:                                [DN_NUMERIC_LITERAL
                                                  lx_numrep = "1"
                                                  sm_exp_type = [DN_INTEGER ^]
                                                  sm_value = 1
                                              ]
1F77D0A_1270B:                                [DN_NUMERIC_LITERAL
                                                  lx_numrep = "2"
                                                  sm_exp_type = [DN_INTEGER ^]
                                                  sm_value = 2
                                              ]
                                          ]
                                      ]
1F77D0A_13920:                        [DN_STM_S
                                          lx_line_count = 9
1F77D0A_1278A:                            [DN_SELECT
                                              lx_line_count = 9
1F77D0A_1380B:                                [DN_SELECT_CLAUSE_S
                                                  lx_line_count = 8
1F77D0A_1282F:                                    [DN_SELECT_CLAUSE
                                                      lx_line_count = 3
                                                      [DN_VOID]
1F77D0A_12F92:                                        [DN_STM_S
                                                          lx_line_count = 3
1F77D0A_128D4:                                            [DN_ACCEPT
                                                              lx_line_count = 3
1F77D0A_12978:                                                [DN_USED_NAME_ID
                                                                  lx_symrep = "ENTRY_A"
                                                                  sm_defn = [DN_ENTRY_ID ^]
                                                              ]
1F77D0A_129EF:                                                [DN_PARAM_S
                                                                  lx_line_count = 0
                                                                  SM_ID_TABLE =
                                                              ]
1F77D0A_12F22:                                                [DN_STM_S
                                                                  lx_line_count = 1
1F77D0A_12A9A:                                                    [DN_ASSIGN
                                                                      lx_line_count = 1
1F77D0A_12B24:                                                        [DN_USED_OBJECT_ID
                                                                          lx_symrep = "RESULT_C"
                                                                          sm_defn = [DN_VAR_ID ^]
                                                                          sm_exp_type = [DN_CONSTRAINED ^]
                                                                          sm_value = Uninitialized
                                                                      ]
1F77D0A_12BC6:                                                        [DN_FUNCTION_CALL
                                                                          sm_exp_type = [DN_INTEGER ^]
                                                                          sm_value = Uninitialized
                                                                          sm_normalized_param_s = [DN_EXP_S ^]
                                                                          lx_prefix = FALSE
1F77D0A_12C85:                                                            [DN_USED_BLTN_OP
                                                                              SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                                                              lx_symrep = "/"
                                                                              sm_operator = INTEGER_DIV
                                                                          ]
```

```
1F77D0A_12EC3:                                        [DN_PARAM_ASSOC_S
1F77D0A_12D38:                                          [DN_USED_OBJECT_ID
                                                          lx_symrep = "RESULT_C"
                                                          sm_defn = [DN_VAR_ID ^]
                                                          sm_exp_type = [DN_CONSTRAINED ^]
                                                          sm_value = Uninitialized
                                                        ]
1F77D0A_12E0F:                                          [DN_NUMERIC_LITERAL
                                                          lx_numrep = "2"
                                                          sm_exp_type = [DN_INTEGER ^]
                                                          sm_value = 2
                                                        ]
                                                      ]
                                                    ]
                                                  ]
                                                ]
                                              ]
                                            ]
                                          ]
1F77D0A_1301D:                            [DN_SELECT_CLAUSE
                                            lx_line_count = 3
                                            [DN_VOID]
1F77D0A_13780:                              [DN_STM_S
                                              lx_line_count = 3
1F77D0A_130C2:                                [DN_ACCEPT
                                                lx_line_count = 3
1F77D0A_13166:                                  [DN_USED_NAME_ID
                                                  lx_symrep = "ENTRY_B"
                                                  sm_defn = [DN_ENTRY_ID ^]
                                                ]
1F77D0A_131DD:                                  [DN_PARAM_S
                                                  lx_line_count = 0
                                                  SM_ID_TABLE =
                                                ]
1F77D0A_13710:                                  [DN_STM_S
                                                  lx_line_count = 1
1F77D0A_13288:                                    [DN_ASSIGN
                                                    lx_line_count = 1
1F77D0A_13312:                                      [DN_USED_OBJECT_ID
                                                        lx_symrep = "RESULT_C"
                                                        sm_defn = [DN_VAR_ID ^]
                                                        sm_exp_type = [DN_CONSTRAINED ^]
                                                        sm_value = Uninitialized
                                                      ]
1F77D0A_133B4:                                      [DN_FUNCTION_CALL
                                                        sm_exp_type = [DN_INTEGER ^]
                                                        sm_value = Uninitialized
                                                        sm_normalized_param_s = [DN_EXP_S ^]
                                                        lx_prefix = FALSE
1F77D0A_13473:                                          [DN_USED_BLTN_OP
                                                          SM_ORIGINAL_NODE = [DN_USED_OP ^]
                                                          lx_symrep = "+"
                                                          sm_operator = INTEGER_ADD
                                                        ]
1F77D0A_136B1:                                          [DN_PARAM_ASSOC_S
1F77D0A_13526:                                            [DN_USED_OBJECT_ID
                                                            lx_symrep = "RESULT_C"
                                                            sm_defn = [DN_VAR_ID ^]
                                                            sm_exp_type = [DN_CONSTRAINED ^]
                                                            sm_value = Uninitialized
                                                          ]
1F77D0A_135FD:                                            [DN_NUMERIC_LITERAL
                                                            lx_numrep = "4"
                                                            sm_exp_type = [DN_INTEGER ^]
                                                            sm_value = 4
                                                          ]
                                                        ]
                                                      ]
                                                    ]
                                                  ]
                                                ]
                                              ]
                                            ]
                                          ]
1F77D0A_1387B:                          [DN_STM_S
                                          lx_line_count = 0
                                        ]
                                      ]
                                    ]
                                  ]
1F77D0A_139AB:                    [DN_ACCEPT
                                    lx_line_count = 1
1F77D0A_13A4F:                      [DN_USED_NAME_ID
                                      lx_symrep = "DONE"
                                      sm_defn = [DN_ENTRY_ID ^]
                                    ]
1F77D0A_13AC6:                      [DN_PARAM_S
                                      lx_line_count = 0
                                      SM_ID_TABLE =
                                    ]
1F77D0A_13B56:                      [DN_STM_S
                                      lx_line_count = 0
                                    ]
                                  ]
```

```
                                                ]
1F77D0A_13C36:                                  [DN_ALTERNATIVE_S
                                                    lx_line_count = 0
                                                ]
                                            ]
                                        ]
                                    ]
1F77D0A_1463D:                      [DN_STM_S
                                        PRE_COMMENT_HEIGHT = 1
                                        lx_line_count = 2
1F77D0A_142F2:                          [DN_ENTRY_CALL
                                            SM_ORIGINAL_NODE = [DN_PROCEDURE_CALL ^]
                                            lx_line_count = 1
                                            sm_normalized_param_s = [DN_EXP_S ^]
1F77D0A_14396:                              [DN_SELECTED
                                                sm_exp_type = null
                                                sm_value = Uninitialized
1F77D0A_1443A:                                  [DN_USED_OBJECT_ID
                                                    lx_symrep = "TASK_C"
                                                    sm_defn = [DN_VAR_ID ^]
                                                    sm_exp_type = [DN_TASK_SPEC ^]
                                                    sm_value = Uninitialized
                                                ]
1F77D0A_144DC:                                  [DN_USED_NAME_ID
                                                    lx_symrep = "DONE"
                                                    sm_defn = [DN_ENTRY_ID ^]
                                                ]
                                            ]
1F77D0A_14553:                              [DN_PARAM_ASSOC_S]
                                        ]
                                    ]
1F77D0A_146F0:                      [DN_ALTERNATIVE_S
                                        lx_line_count = 0
                                    ]
                                ]
                            ]
1F77D0A_1483C:              [DN_PRAGMA_S
                                lx_line_count = 0
                            ]
                        ]
```

Appendix D

Experiment Output

```
main
1000 Iterations of simple tasking system takes    11.8167 seconds.
main
1000 Iterations of simple tasking system takes    11.9333 seconds.
main
1000 Iterations of simple tasking system takes    11.9833 seconds.
main
1000 Iterations of simple tasking system takes    12.0500 seconds.
main
1000 Iterations of simple tasking system takes    14.3834 seconds.
main
1000 Iterations of simple tasking system takes    14.3833 seconds.
main
1000 Iterations of simple tasking system takes    14.4000 seconds.
main
1000 Iterations of simple tasking system takes    14.3833 seconds.
main
1000 Iterations of simple tasking system takes    14.3834 seconds.
main
1000 Iterations of simple tasking system takes    14.3833 seconds.
main
1000 Iterations of simple tasking system takes    14.3834 seconds.
main
1000 Iterations of simple tasking system takes    11.7333 seconds.
main
1000 Iterations of simple tasking system takes    14.5667 seconds.
main
1000 Iterations of simple tasking system takes    14.5667 seconds.
main
1000 Iterations of simple tasking system takes    14.5667 seconds.
main
1000 Iterations of simple tasking system takes    14.5667 seconds.
main
1000 Iterations of simple tasking system takes    14.5667 seconds.
```

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

 Number of sample values is          7000
 Clock Resolution            =        1.000000000000000 seconds.
 Clock Resolution (average)  =        1.000000000000000 seconds.
 Clock Resolution (variance) =        0.000000000000000 seconds.


Test Name:   A000091              Class Name:  Composite
          1.2000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark


Test Name:   A000092                          Class Name: composite

Average time per cycle :  2784.33 milliseconds

Average Whetstone rating :  359 KWIPS


Iteration  1
 73717.0000 73717.0000      0.0000
Test Iteration  1
 73717.0000 73717.0000      0.0000
Iteration  2
 73718.0000 73718.0000      0.0000
Test Iteration  2
 73718.0000 73718.0000      0.0000
Iteration  4
 73718.0000 73718.0000      0.0000
Test Iteration  4
 73719.0000 73719.0000      0.0000
Iteration  8
 73720.0000 73720.0000      0.0000
Test Iteration  8
 73720.0000 73721.0000      1.0000
Iteration  16
 73721.0000 73721.0000      0.0000
Test Iteration  16
 73721.0000 73723.0000      2.0000
Iteration  32
 73723.0000 73723.0000      0.0000
Test Iteration  32
 73723.0000 73727.0000      4.0000
Iteration  64
 73727.0000 73727.0000      0.0000
Test Iteration  64
 73727.0000 73734.0000      7.0000
Iteration  128
 73734.0000 73735.0000      1.0000
Test Iteration  128
 73735.0000 73749.0000     14.0000
Iteration  256
 73749.0000 73750.0000      1.0000
Test Iteration  256
 73750.0000 73777.0000     27.0000
Iteration  512
 73777.0000 73780.0000      3.0000
Test Iteration  512
 73780.0000 73834.0000     54.0000
Iteration  1024
 73834.0000 73839.0000      5.0000
Test Iteration  1024
 73839.0000 73947.0000    108.0000

Test Name:   C000001                     Class Name:  Tasking
CPU Time:      1005.9  microseconds
Wall Time:     1005.9  microseconds.     Iteration Count:    1024
Test Description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop,


Iteration  1
 73949.0000 73949.0000      0.0000
Test Iteration  1
 73949.0000 73950.0000      1.0000
Iteration  2
 73950.0000 73950.0000      0.0000
Test Iteration  2
 73950.0000 73950.0000      0.0000
Iteration  4
 73950.0000 73951.0000      1.0000
Test Iteration  4
 73951.0000 73951.0000      0.0000
Iteration  8
 73951.0000 73951.0000      0.0000
Test Iteration  8
 73952.0000 73953.0000      1.0000
Iteration  16
 73953.0000 73953.0000      0.0000
Test Iteration  16
 73953.0000 73955.0000      2.0000
Iteration  32
 73955.0000 73955.0000      0.0000
Test Iteration  32
 73955.0000 73959.0000      4.0000
Iteration  64
 73959.0000 73960.0000      1.0000
Test Iteration  64
 73960.0000 73967.0000      7.0000
Iteration  128
 73967.0000 73968.0000      1.0000
Test Iteration  128
 73968.0000 73982.0000     14.0000
Iteration  256
 73982.0000 73984.0000      2.0000
Test Iteration  256
 73984.0000 74012.0000     28.0000
Iteration  512
 74012.0000 74015.0000      3.0000
Test Iteration  512
 74015.0000 74071.0000     56.0000
Iteration  1024
 74072.0000 74077.0000      5.0000
Test Iteration  1024
 74077.0000 74190.0000    113.0000

Test Name:   C000002                     Class Name:  Tasking
CPU Time:      1054.7  microseconds
Wall Time:     1054.7  microseconds.     Iteration Count:    1024
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop

Iteration  1
 74557.0000 74557.0000      0.0000
Test Iteration  1
 74558.0000 74558.0000      0.0000
Iteration  2
 74558.0000 74558.0000      0.0000
Test Iteration  2
 74559.0000 74559.0000      0.0000
Iteration  4
 74559.0000 74559.0000      0.0000
Test Iteration  4
 74559.0000 74559.0000      0.0000
Iteration  8
 74560.0000 74560.0000      0.0000
Test Iteration  8
 74560.0000 74560.0000      0.0000
Iteration  16
 74560.0000 74561.0000      1.0000
Test Iteration  16
 74561.0000 74562.0000      1.0000
Iteration  32
 74562.0000 74562.0000      0.0000
Test Iteration  32
 74562.0000 74564.0000      2.0000
Iteration  64
 74564.0000 74565.0000      1.0000
Test Iteration  64
 74565.0000 74568.0000      3.0000
Iteration  128
 74569.0000 74569.0000      0.0000
Test Iteration  128
 74570.0000 74576.0000      6.0000
Iteration  256
 74576.0000 74578.0000      2.0000
Test Iteration  256
 74578.0000 74591.0000     13.0000
Iteration  512
 74591.0000 74594.0000      3.0000
Test Iteration  512
 74594.0000 74620.0000     26.0000
Iteration  1024
 74620.0000 74626.0000      6.0000
Test Iteration  1024
 74626.0000 74678.0000     52.0000
Iteration  2048
 74678.0000 74689.0000     11.0000
Test Iteration  2048
 74690.0000 74793.0000    103.0000

Test Name:   H000004                              Class Name:  Chapter 13
CPU Time:       449.2  microseconds
Wall Time:      449.2  microseconds.          Iteration Count:    2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
 For this test the arrays are NOT PACKED with the pragma 'PACK.'
 For this test the operations are performed on components in a loop.


Iteration  1
 75295.0000 75295.0000      0.0000
Test Iteration  1
 75296.0000 75296.0000      0.0000
Iteration  2
 75296.0000 75296.0000      0.0000
Test Iteration  2
 75296.0000 75296.0000      0.0000
Iteration  4
 75297.0000 75297.0000      0.0000
Test Iteration  4
 75297.0000 75297.0000      0.0000
Iteration  8
 75297.0000 75297.0000      0.0000
Test Iteration  8
 75297.0000 75297.0000      0.0000
Iteration  16
 75298.0000 75298.0000      0.0000
Test Iteration  16
 75298.0000 75298.0000      0.0000
Iteration  32
 75298.0000 75298.0000      0.0000
Test Iteration  32
 75299.0000 75299.0000      0.0000
Iteration  64
 75299.0000 75299.0000      0.0000
Test Iteration  64
 75300.0000 75300.0000      0.0000
Iteration  128
 75300.0000 75301.0000      1.0000
Test Iteration  128
 75301.0000 75302.0000      1.0000
Iteration  256
 75302.0000 75303.0000      1.0000
Test Iteration  256
 75304.0000 75305.0000      1.0000
Iteration  512
 75305.0000 75308.0000      3.0000
Test Iteration  512
 75308.0000 75311.0000      3.0000
Iteration  1024
 75311.0000 75316.0000      5.0000
Test Iteration  1024
 75317.0000 75322.0000      5.0000
Iteration  2048
 75323.0000 75333.0000     10.0000
Test Iteration  2048
 75333.0000 75345.0000     12.0000
Iteration  4096
 75345.0000 75366.0000     21.0000
Test Iteration  4096
 75366.0000 75390.0000     24.0000
Iteration  8192
 75390.0000 75431.0000     41.0000
Test Iteration  8192
 75431.0000 75479.0000     48.0000
Iteration  16384
 75479.0000 75561.0000     82.0000
Test Iteration  16384
 75561.0000 75657.0000     96.0000
***** INCOMPLETE MEASUREMENT *****

Test Name:   P000001                              Class Name:  Procedure
CPU Time:         8.5  microseconds
Wall Time:        8.5  microseconds.          Iteration Count:   16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters


Iteration  1
 75659.0000 75659.0000      0.0000
```

```
Test Iteration  1
 75659.0000 75659.0000      0.0000
Iteration  2
 75660.0000 75660.0000      0.0000
Test Iteration  2
 75660.0000 75660.0000      0.0000
Iteration  4
 75660.0000 75660.0000      0.0000
Test Iteration  4
 75660.0000 75660.0000      0.0000
Iteration  8
 75661.0000 75661.0000      0.0000
Test Iteration  8
 75661.0000 75661.0000      0.0000
Iteration  16
 75661.0000 75661.0000      0.0000
Test Iteration  16
 75662.0000 75662.0000      0.0000
Iteration  32
 75662.0000 75662.0000      0.0000
Test Iteration  32
 75663.0000 75663.0000      0.0000
Iteration  64
 75663.0000 75663.0000      0.0000
Test Iteration  64
 75664.0000 75664.0000      0.0000
Iteration  128
 75665.0000 75666.0000      1.0000
Test Iteration  128
 75666.0000 75667.0000      1.0000
Iteration  256
 75667.0000 75669.0000      2.0000
Test Iteration  256
 75669.0000 75671.0000      2.0000
Iteration  512
 75671.0000 75675.0000      4.0000
Test Iteration  512
 75675.0000 75679.0000      4.0000
Iteration  1024
 75679.0000 75685.0000      6.0000
Test Iteration  1024
 75686.0000 75694.0000      8.0000
Iteration  2048
 75694.0000 75707.0000     13.0000
Test Iteration  2048
 75707.0000 75724.0000     17.0000
Iteration  4096
 75724.0000 75749.0000     25.0000
Test Iteration  4096
 75749.0000 75783.0000     34.0000
Iteration  8192
 75784.0000 75833.0000     49.0000
Test Iteration  8192
 75833.0000 75902.0000     69.0000
Iteration  16384
 75902.0000 76001.0000     99.0000
Test Iteration  16384
 76001.0000 76138.0000    137.0000

Test Name:    P000010                          Class Name:  Procedure
CPU Time:        23.2  microseconds
Wall Time:       23.2  microseconds.           Iteration Count:   16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
 76140.0000 76140.0000      0.0000
Test Iteration  1
 76140.0000 76140.0000      0.0000
Iteration  2
 76140.0000 76141.0000      1.0000
Test Iteration  2
 76141.0000 76141.0000      0.0000
Iteration  4
 76141.0000 76141.0000      0.0000
Test Iteration  4
 76141.0000 76141.0000      0.0000
Iteration  8
 76142.0000 76142.0000      0.0000
Test Iteration  8
 76142.0000 76143.0000      1.0000
Iteration  16
 76143.0000 76143.0000      0.0000
Test Iteration  16
 76143.0000 76144.0000      1.0000
Iteration  32
 76144.0000 76144.0000      0.0000
Test Iteration  32
 76145.0000 76146.0000      1.0000
Iteration  64
 76146.0000 76147.0000      1.0000
Test Iteration  64
 76147.0000 76150.0000      3.0000
Iteration  128
 76150.0000 76151.0000      1.0000
Test Iteration  128
 76151.0000 76158.0000      7.0000
Iteration  256
 76158.0000 76159.0000      1.0000
Test Iteration  256
 76160.0000 76173.0000     13.0000
Iteration  512
 76173.0000 76175.0000      2.0000
Test Iteration  512
 76176.0000 76202.0000     26.0000
Iteration  1024
 76202.0000 76207.0000      5.0000
Test Iteration  1024
 76207.0000 76259.0000     52.0000
Iteration  2048
 76259.0000 76270.0000     11.0000
Test Iteration  2048
 76270.0000 76374.0000    104.0000

Test Name:    T000001                          Class Name:  Tasking
CPU Time:       454.1  microseconds
Wall Time:      454.1  microseconds.           Iteration Count:   2048
Test Description:
 Minimum rendezvous, entry call and return time
 1 task 1 entry , task inside procedure
 no select


Iteration  1
 76376.0000 76376.0000      0.0000
Test Iteration  1
 76376.0000 76376.0000      0.0000
Iteration  2
 76377.0000 76377.0000      0.0000
Test Iteration  2
 76377.0000 76377.0000      0.0000
Iteration  4
 76377.0000 76377.0000      0.0000
Test Iteration  4
 76378.0000 76378.0000      0.0000
Iteration  8
 76378.0000 76378.0000      0.0000
Test Iteration  8
 76379.0000 76380.0000      1.0000
Iteration  16
 76380.0000 76380.0000      0.0000
Test Iteration  16
 76380.0000 76382.0000      2.0000
Iteration  32
 76382.0000 76383.0000      1.0000
Test Iteration  32
 76383.0000 76387.0000      4.0000
Iteration  64
 76387.0000 76387.0000      0.0000
Test Iteration  64
 76387.0000 76395.0000      8.0000
Iteration  128
 76395.0000 76396.0000      1.0000
Test Iteration  128
 76396.0000 76411.0000     15.0000
Iteration  256
 76411.0000 76413.0000      2.0000
Test Iteration  256
 76413.0000 76443.0000     30.0000
Iteration  512
 76443.0000 76446.0000      3.0000
Test Iteration  512
 76446.0000 76507.0000     61.0000
Iteration  1024
 76507.0000 76512.0000      5.0000
Test Iteration  1024
 76513.0000 76634.0000    121.0000

Test Name:    T000004                          Class Name:  Tasking
CPU Time:       566.4  microseconds
Wall Time:      566.4  microseconds.           Iteration Count:   1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement


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

 Number of sample values is          7000
 Clock Resolution              =        1.000000000000000 seconds.
 Clock Resolution (average)    =        1.000000000000000 seconds.
 Clock Resolution (variance)   =        0.000000000000000 seconds.


Test Name:    A000091                Class Name:  Composite
          1.2000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark

Test Name:    A000092                          Class Name: composite

Average time per cycle :  2834.67 milliseconds

Average Whetstone rating :  353 KWIPS


Iteration  1
 77938.0000 77938.0000      0.0000
Test Iteration  1
 77938.0000 77939.0000      1.0000
Iteration  2
 77939.0000 77939.0000      0.0000
Test Iteration  2
 77939.0000 77939.0000      0.0000
Iteration  4
 77939.0000 77939.0000      0.0000
Test Iteration  4
 77940.0000 77940.0000      0.0000
Iteration  8
 77941.0000 77941.0000      0.0000
Test Iteration  8
 77941.0000 77942.0000      1.0000
Iteration  16
 77942.0000 77942.0000      0.0000
Test Iteration  16
 77942.0000 77944.0000      2.0000
Iteration  32
 77944.0000 77944.0000      0.0000
Test Iteration  32
 77945.0000 77948.0000      3.0000
Iteration  64
 77948.0000 77948.0000      0.0000
Test Iteration  64
 77949.0000 77955.0000      6.0000
Iteration  128
 77956.0000 77956.0000      0.0000
Test Iteration  128
 77957.0000 77970.0000     13.0000
Iteration  256
 77970.0000 77972.0000      2.0000
Test Iteration  256
 77972.0000 77999.0000     27.0000
Iteration  512
 77999.0000 78002.0000      3.0000
Test Iteration  512
 78002.0000 78056.0000     54.0000
Iteration  1024
 78056.0000 78062.0000      6.0000
Test Iteration  1024
 78062.0000 78170.0000    108.0000

Test Name:    C000001                          Class Name:  Tasking
CPU Time:       996.1  microseconds
Wall Time:      996.1  microseconds.           Iteration Count:   1024
Test Description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop,


Iteration  1
 78173.0000 78173.0000      0.0000
Test Iteration  1
 78173.0000 78173.0000      0.0000
Iteration  2
```

```
 78173.0000 78173.0000     0.0000
Test Iteration  2
 78173.0000 78174.0000     1.0000
Iteration  4
 78174.0000 78174.0000     0.0000
Test Iteration  4
 78174.0000 78175.0000     1.0000
Iteration  8
 78175.0000 78175.0000     0.0000
Test Iteration  8
 78175.0000 78176.0000     1.0000
Iteration  16
 78176.0000 78176.0000     0.0000
Test Iteration  16
 78176.0000 78178.0000     2.0000
Iteration  32
 78179.0000 78179.0000     0.0000
Test Iteration  32
 78179.0000 78182.0000     3.0000
Iteration  64
 78183.0000 78183.0000     0.0000
Test Iteration  64
 78183.0000 78190.0000     7.0000
Iteration  128
 78191.0000 78191.0000     0.0000
Test Iteration  128
 78191.0000 78206.0000    15.0000
Iteration  256
 78206.0000 78207.0000     1.0000
Test Iteration  256
 78207.0000 78236.0000    29.0000
Iteration  512
 78236.0000 78239.0000     3.0000
Test Iteration  512
 78239.0000 78296.0000    57.0000
Iteration  1024
 78296.0000 78302.0000     6.0000
Test Iteration  1024
 78302.0000 78415.0000   113.0000

Test Name:   C000002                          Class Name:  Tasking
CPU Time:       1044.9  microseconds
Wall Time:      1044.9  microseconds.         Iteration Count:   1024
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop


Iteration  1
 78783.0000 78783.0000     0.0000
Test Iteration  1
 78783.0000 78783.0000     0.0000
Iteration  2
 78784.0000 78784.0000     0.0000
Test Iteration  2
 78784.0000 78784.0000     0.0000
Iteration  4
 78784.0000 78784.0000     0.0000
Test Iteration  4
 78784.0000 78785.0000     1.0000
Iteration  8
 78785.0000 78785.0000     0.0000
Test Iteration  8
 78785.0000 78785.0000     0.0000
Iteration  16
 78786.0000 78786.0000     0.0000
Test Iteration  16
 78786.0000 78787.0000     1.0000
Iteration  32
 78787.0000 78787.0000     0.0000
Test Iteration  32
 78787.0000 78789.0000     2.0000
Iteration  64
 78789.0000 78790.0000     1.0000
Test Iteration  64
 78790.0000 78793.0000     3.0000
Iteration  128
 78794.0000 78795.0000     1.0000
Test Iteration  128
 78795.0000 78801.0000     6.0000
Iteration  256
 78802.0000 78803.0000     1.0000
Test Iteration  256
 78803.0000 78816.0000    13.0000
Iteration  512
 78816.0000 78819.0000     3.0000
Test Iteration  512
 78819.0000 78845.0000    26.0000
Iteration  1024
 78845.0000 78851.0000     6.0000
Test Iteration  1024
 78851.0000 78903.0000    52.0000
Iteration  2048
 78903.0000 78914.0000    11.0000
Test Iteration  2048
 78915.0000 79018.0000   103.0000

Test Name:   H000004                          Class Name:  Chapter 13
CPU Time:        449.2  microseconds
Wall Time:       449.2  microseconds.         Iteration Count:   2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
 For this test the arrays are NOT PACKED with the pragma 'PACK.'
 For this test the operations are performed on components in a loop.


Iteration  1
 79584.0000 79584.0000     0.0000
Test Iteration  1
 79584.0000 79584.0000     0.0000
Iteration  2
 79584.0000 79584.0000     0.0000
Test Iteration  2
 79584.0000 79584.0000     0.0000
Iteration  4
 79585.0000 79585.0000     0.0000
Test Iteration  4
 79585.0000 79585.0000     0.0000
Iteration  8
 79585.0000 79585.0000     0.0000
Test Iteration  8
 79586.0000 79586.0000     0.0000
Iteration  16
 79586.0000 79586.0000     0.0000
Test Iteration  16
 79586.0000 79586.0000     0.0000
Iteration  32
 79587.0000 79587.0000     0.0000
Test Iteration  32
 79587.0000 79587.0000     0.0000
Iteration  64
 79587.0000 79588.0000     1.0000
Test Iteration  64
 79588.0000 79588.0000     0.0000
Iteration  128
 79589.0000 79589.0000     0.0000
Test Iteration  128
 79589.0000 79590.0000     1.0000
Iteration  256
 79591.0000 79592.0000     1.0000
Test Iteration  256
 79592.0000 79594.0000     2.0000
Iteration  512
 79594.0000 79597.0000     3.0000
Test Iteration  512
 79597.0000 79600.0000     3.0000
Iteration  1024
 79601.0000 79606.0000     5.0000
Test Iteration  1024
 79607.0000 79613.0000     6.0000
Iteration  2048
 79613.0000 79625.0000    12.0000
Test Iteration  2048
 79625.0000 79638.0000    13.0000
Iteration  4096
 79638.0000 79661.0000    23.0000
Test Iteration  4096
 79662.0000 79688.0000    26.0000
Iteration  8192
 79688.0000 79734.0000    46.0000
Test Iteration  8192
 79735.0000 79787.0000    52.0000
Iteration  16384
 79787.0000 79879.0000    92.0000
Test Iteration  16384
 79880.0000 79984.0000   104.0000

Test Name:   P000001                          Class Name:  Procedure
CPU Time:          7.3  microseconds
Wall Time:         7.3  microseconds.         Iteration Count:   16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters


Iteration  1
 79986.0000 79986.0000     0.0000
Test Iteration  1
 79986.0000 79986.0000     0.0000
Iteration  2
 79986.0000 79986.0000     0.0000
Test Iteration  2
 79987.0000 79987.0000     0.0000
Iteration  4
 79987.0000 79987.0000     0.0000
Test Iteration  4
 79987.0000 79987.0000     0.0000
Iteration  8
 79987.0000 79988.0000     1.0000
Test Iteration  8
 79988.0000 79988.0000     0.0000
Iteration  16
 79988.0000 79988.0000     0.0000
Test Iteration  16
 79988.0000 79988.0000     0.0000
Iteration  32
 79989.0000 79989.0000     0.0000
Test Iteration  32
 79990.0000 79990.0000     0.0000
Iteration  64
 79990.0000 79991.0000     1.0000
Test Iteration  64
 79991.0000 79991.0000     0.0000
Iteration  128
 79992.0000 79993.0000     1.0000
Test Iteration  128
 79993.0000 79994.0000     1.0000
Iteration  256
 79994.0000 79996.0000     2.0000
Test Iteration  256
 79996.0000 79999.0000     3.0000
Iteration  512
 79999.0000 80002.0000     3.0000
Test Iteration  512
 80002.0000 80007.0000     5.0000
Iteration  1024
 80007.0000 80013.0000     6.0000
Test Iteration  1024
 80014.0000 80022.0000     8.0000
Iteration  2048
 80023.0000 80036.0000    13.0000
Test Iteration  2048
 80036.0000 80053.0000    17.0000
Iteration  4096
 80054.0000 80080.0000    26.0000
Test Iteration  4096
 80080.0000 80115.0000    35.0000
Iteration  8192
 80115.0000 80168.0000    53.0000
Test Iteration  8192
 80168.0000 80238.0000    70.0000
Iteration  16384
 80238.0000 80343.0000   105.0000
Test Iteration  16384
 80343.0000 80482.0000   139.0000

Test Name:   P000010                          Class Name:  Procedure
CPU Time:         20.8  microseconds
Wall Time:        20.8  microseconds.         Iteration Count:   16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
 80485.0000 80485.0000     0.0000
Test Iteration  1
 80485.0000 80485.0000     0.0000
Iteration  2
 80485.0000 80485.0000     0.0000
Test Iteration  2
 80486.0000 80486.0000     0.0000
Iteration  4
 80486.0000 80486.0000     0.0000
Test Iteration  4
 80486.0000 80486.0000     0.0000
Iteration  8
 80486.0000 80487.0000     1.0000
Test Iteration  8
 80487.0000 80487.0000     0.0000
Iteration  16
```

```
 80487.0000 80488.0000      1.0000
Test Iteration  16
 80488.0000 80489.0000      1.0000
Iteration  32
 80489.0000 80489.0000      0.0000
Test Iteration  32
 80489.0000 80491.0000      2.0000
Iteration  64
 80491.0000 80491.0000      0.0000
Test Iteration  64
 80492.0000 80495.0000      3.0000
Iteration  128
 80495.0000 80496.0000      1.0000
Test Iteration  128
 80496.0000 80502.0000      6.0000
Iteration  256
 80502.0000 80504.0000      2.0000
Test Iteration  256
 80504.0000 80517.0000     13.0000
Iteration  512
 80517.0000 80520.0000      3.0000
Test Iteration  512
 80520.0000 80545.0000     25.0000
Iteration  1024
 80546.0000 80551.0000      5.0000
Test Iteration  1024
 80551.0000 80602.0000     51.0000
Iteration  2048
 80602.0000 80613.0000     11.0000
Test Iteration  2048
 80614.0000 80715.0000    101.0000

Test Name:    T000001                          Class Name:  Tasking
CPU Time:        439.5  microseconds
Wall Time:       439.5  microseconds.          Iteration Count:   2048
Test Description:
 Minimum rendezvous, entry call and return time
  1 task 1 entry , task inside procedure
  no select


Iteration  1
 80718.0000 80718.0000      0.0000
Test Iteration  1
 80718.0000 80718.0000      0.0000
Iteration  2
 80718.0000 80718.0000      0.0000
Test Iteration  2
 [illegible]
Iteration  4
 80719.0000 80719.0000      0.0000
Test Iteration  4
 80719.0000 80720.0000      1.0000
Iteration  8
 80720.0000 80720.0000      0.0000
Test Iteration  8
 80720.0000 80721.0000      1.0000
Iteration  16
 80722.0000 80722.0000      0.0000
Test Iteration  16
 80722.0000 80724.0000      2.0000
Iteration  32
 80724.0000 80724.0000      0.0000
Test Iteration  32
 80724.0000 80728.0000      4.0000
Iteration  64
 80728.0000 80729.0000      1.0000
Test Iteration  64
 80729.0000 80736.0000      7.0000
Iteration  128
 80737.0000 80737.0000      0.0000
Test Iteration  128
 80738.0000 80752.0000     14.0000
Iteration  256
 80753.0000 80754.0000      1.0000
Test Iteration  256
 80754.0000 80784.0000     30.0000
Iteration  512
 80784.0000 80787.0000      3.0000
Test Iteration  512
 80787.0000 80847.0000     60.0000
Iteration  1024
 80847.0000 80853.0000      6.0000
Test Iteration  1024
 80853.0000 80972.0000    119.0000

Test Name:    T000004                          Class Name:  Tasking
CPU Time:        551.8  microseconds
Wall Time:       551.8  microseconds.          Iteration Count:   1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement


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

 Number of sample values is          7000
 Clock Resolution              =       1.000000000000000 seconds.
 Clock Resolution (average)    =       1.000000000000000 seconds.
 Clock Resolution (variance)   =       0.000000000000000 seconds.


Test Name:    A000091              Class Name:  Composite
           1.2000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark


Test Name:    A000092                        Class Name: composite

Average time per cycle :  2845.33 milliseconds

Average Whetstone rating :  351 KWIPS


Iteration  1
 37482.0000 37482.0000      0.0000
Iteration  2
 37483.0000 37483.0000      0.0000
Iteration  4
 37483.0000 37483.0000      0.0000
Iteration  8
 37484.0000 37484.0000      0.0000
Iteration  16
 37485.0000 37485.0000      0.0000
Iteration  32
 37487.0000 37488.0000      1.0000
Iteration  64
 37491.0000 37492.0000      1.0000
Iteration  128
 37499.0000 37500.0000      1.0000
Iteration  256
 37514.0000 37515.0000      1.0000
Iteration  512
 37543.0000 37546.0000      3.0000
Iteration  1024
 37600.0000 37606.0000      6.0000

Test Name:    C000001                          Class Name:  Tasking
CPU Time:       1005.9  microseconds
Wall Time:      1005.9  microseconds.          Iteration Count:   1024
Test Description:
 Task create and terminate measurement
  with one task, no entries, when task is in a procedure
  using a task type in a package, no select statement, no loop,


Iteration  1
 37718.0000 37718.0000      0.0000
Iteration  2
 37718.0000 37718.0000      0.0000
Iteration  4
 37719.0000 37719.0000      0.0000
Iteration  8
 37719.0000 37719.0000      0.0000
Iteration  16
 37720.0000 37720.0000      0.0000
Iteration  32
 37723.0000 37723.0000      0.0000
Iteration  64
 37726.0000 37727.0000      1.0000
Iteration  128
 37734.0000 37735.0000      1.0000
Iteration  256
 37749.0000 37751.0000      2.0000
Iteration  512
 37779.0000 37782.0000      3.0000
Iteration  1024
 37839.0000 37845.0000      6.0000

Test Name:    C000002                          Class Name:  Tasking
CPU Time:       1054.7  microseconds
Wall Time:      1054.7  microseconds.          Iteration Count:   1024
Test Description:
 Task create and terminate time measurement.
  with one task, no entries when task is in a procedure,
  task defined and used in procedure, no select statement, no loop



Iteration  1
 38327.0000 38327.0000      0.0000
Iteration  2
 38328.0000 38328.0000      0.0000
Iteration  4
 38328.0000 38328.0000      0.0000
Iteration  8
 38328.0000 38329.0000      1.0000
Iteration  16
 38329.0000 38329.0000      0.0000
Iteration  32
 38330.0000 38330.0000      0.0000
Iteration  64
 38332.0000 38333.0000      1.0000
Iteration  128
 38336.0000 38337.0000      1.0000
Iteration  256
 38343.0000 38345.0000      2.0000
Iteration  512
 38358.0000 38361.0000      3.0000
Iteration  1024
 38387.0000 38392.0000      5.0000
Iteration  2048
 38444.0000 38455.0000     11.0000

Test Name:    H000004                          Class Name:  Chapter 13
CPU Time:        449.2  microseconds
Wall Time:       449.2  microseconds.          Iteration Count:   2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
  For this test the arrays are NOT PACKED with the pragma 'PACK.'
  For this test the operations are performed on components in a loop.


Iteration  1
 39118.0000 39118.0000      0.0000
Iteration  2
 39119.0000 39119.0000      0.0000
Iteration  4
 39119.0000 39119.0000      0.0000
Iteration  8
 39119.0000 39119.0000      0.0000
Iteration  16
 39119.0000 39120.0000      1.0000
Iteration  32
 39120.0000 39120.0000      0.0000
Iteration  64
 39120.0000 39121.0000      1.0000
Iteration  128
 39121.0000 39122.0000      1.0000
Iteration  256
 39123.0000 39125.0000      2.0000
Iteration  512
 39126.0000 39129.0000      3.0000
Iteration  1024
 39133.0000 39138.0000      5.0000
Iteration  2048
 39145.0000 39156.0000     11.0000
Iteration  4096
 39169.0000 39192.0000     23.0000
Iteration  8192
 39217.0000 39262.0000     45.0000
Iteration  16384
 39313.0000 39403.0000     90.0000

Test Name:    P000001                          Class Name:  Procedure
CPU Time:          7.9  microseconds
Wall Time:         7.9  microseconds.          Iteration Count:  16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters


Iteration  1
 39508.0000 39508.0000      0.0000
Iteration  2
 39509.0000 39509.0000      0.0000
Iteration  4
```

```
39509.0000 39509.0000     0.0000
Iteration  8
 39509.0000 39509.0000     0.0000
Iteration  16
 39510.0000 39510.0000     0.0000
Iteration  32
 39510.0000 39510.0000     0.0000
Iteration  64
 39511.0000 39511.0000     0.0000
Iteration  128
 39512.0000 39513.0000     1.0000
Iteration  256
 39514.0000 39515.0000     1.0000
Iteration  512
 39518.0000 39521.0000     3.0000
Iteration  1024
 39525.0000 39532.0000     7.0000
Iteration  2048
 39540.0000 39553.0000    13.0000
Iteration  4096
 39569.0000 39594.0000    25.0000
Iteration  8192
 39627.0000 39678.0000    51.0000
Iteration  16384
 39742.0000 39844.0000   102.0000

Test Name:   P000010                    Class Name:  Procedure
CPU Time:        15.9  microseconds
Wall Time:       16.5  microseconds.    Iteration Count:  16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
 39976.0000 39976.0000     0.0000
Iteration  2
 39976.0000 39976.0000     0.0000
Iteration  4
 39976.0000 39976.0000     0.0000
Iteration  8
 39977.0000 39977.0000     0.0000
Iteration  16
 39978.0000 39978.0000     0.0000
Iteration  32
 39979.0000 39979.0000     0.0000
Iteration  64
 39981.0000 39981.0000     0.0000
Iteration  128
 39985.0000 39985.0000     0.0000
Iteration  256
 39992.0000 39993.0000     1.0000
Iteration  512
 40007.0000 40009.0000     2.0000
Iteration  1024
 40035.0000 40041.0000     6.0000
Iteration  2048
 40093.0000 40104.0000    11.0000

Test Name:   T000001                    Class Name:  Tasking
CPU Time:       454.1  microseconds
Wall Time:      454.1  microseconds.    Iteration Count:  2048
Test Description:
 Minimum rendezvous, entry call and return time
 1 task 1 entry , task inside procedure
 no select


Iteration  1
 40210.0000 40210.0000     0.0000
Iteration  2
 40210.0000 40210.0000     0.0000
Iteration  4
 40211.0000 40211.0000     0.0000
Iteration  8
 40212.0000 40212.0000     0.0000
Iteration  16
 40213.0000 40213.0000     0.0000
Iteration  32
 40215.0000 40215.0000     0.0000
Iteration  64
 40219.0000 40220.0000     1.0000
Iteration  128
 40228.0000 40228.0000     0.0000
Iteration  256
 40244.0000 40245.0000     1.0000
Iteration  512
 40276.0000 40279.0000     3.0000
Iteration  1024
 40340.0000 40346.0000     6.0000

Test Name:   T000004                    Class Name:  Tasking
CPU Time:       571.3  microseconds
Wall Time:      571.3  microseconds.    Iteration Count:  1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement


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

 Number of sample values is        7000
 Clock Resolution              =      1.000000000000000 seconds.
 Clock Resolution (average)    =      1.000000000000000 seconds.
 Clock Resolution (variance)   =      0.000000000000000 seconds.


Test Name:   A000091            Class Name:  Composite
            1.2000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark


Test Name:   A000092                        Class Name: composite

Average time per cycle :  2804.00 milliseconds

Average Whetstone rating :  357 KWIPS


Iteration  1
 77817.0000 77817.0000     0.0000
Iteration  2
 77818.0000 77818.0000     0.0000
Iteration  4
 77818.0000 77818.0000     0.0000
Iteration  8
 77819.0000 77819.0000     0.0000
Iteration  16
 77820.0000 77820.0000     0.0000
Iteration  32
 77822.0000 77822.0000     0.0000
Iteration  64
 77826.0000 77826.0000     0.0000
Iteration  128
 77833.0000 77833.0000     0.0000
Iteration  256
 77847.0000 77848.0000     1.0000
Iteration  512
 77875.0000 77878.0000     3.0000
Iteration  1024
 77932.0000 77937.0000     5.0000

Test Name:   C000001                            Class Name:  Tasking
CPU Time:       996.1  microseconds
Wall Time:      996.1  microseconds.            Iteration Count:  1024
Test Description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop,


Iteration  1
 78046.0000 78046.0000     0.0000
Iteration  2
 78047.0000 78047.0000     0.0000
Iteration  4
 78047.0000 78047.?000     0.0000
Iteration  8
 78048.0000 78048.0000     0.0000
Iteration  16
 78049.0000 78049.0000     0.0000
Iteration  32
 78051.0000 78051.0000     0.0000
Iteration  64
 78055.0000 78056.0000     1.0000
Iteration  128
 78063.0000 78064.0000     1.0000
Iteration  256
 78078.0000 78080.0000     2.0000
Iteration  512
 78108.0000 78111.0000     3.0000
 . eration  1024
 ?8167.0000 78172.0000     5.0000

Test Name:   C000002                            Class Name:  Tasking
CPU Time:      1054.7  microseconds
Wall Time:     1054.7  microseconds.            Iteration Count:  1024
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop


Iteration  1
 78651.0000 78651.0000     0.0000
Iteration  2
 78651.0000 78651.0000     0.0000
Iteration  4
 78652.0000 78652.0000     0.0000
Iteration  8
 78653.0000 78653.0000     0.0000
Iteration  16
 78653.0000 78654.0000     1.0000
Iteration  32
 78655.0000 78655.0000     0.0000
Iteration  64
 78657.0000 78657.0000     0.0000
Iteration  128
 78660.0000 78661.0000     1.0000
Iteration  256
 78668.0000 78669.0000     1.0000
Iteration  512
 78683.0000 78685.0000     2.0000
Iteration  1024
 78712.0000 78718.0000     6.0000
Iteration  2048
 78770.0000 78781.0000    11.0000

Test Name:   H000004                            Class Name:  Chapter 13
CPU Time:       449.2  microseconds
Wall Time:      449.2  microseconds.            Iteration Count:  2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
 For this test the arrays are NOT PACKED with the pragma 'PACK.'
 For this test the operations are performed on components in a loop.

Iteration  1
 79400.0000 79400.0000     0.0000
Iteration  2
 79400.0000 79400.0000     0.0000
Iteration  4
 79401.0000 79401.0000     0.0000
Iteration  8
 79401.0000 79401.0000     0.0000
Iteration  16
 79401.0000 79401.0000     0.0000
Iteration  32
 79402.0000 79402.0000     0.0000
Iteration  64
 79402.0000 79402.0000     0.0000
Iteration  128
 79403.0000 79404.0000     1.0000
Iteration  256
 79405.0000 79406.0000     1.0000
Iteration  512
 79408.0000 79410.0000     2.0000
Iteration  1024
 79414.0000 79419.0000     5.0000
Iteration  2048
 79425.0000 79436.0000    11.0000
Iteration  4096
 79448.0000 79469.0000    21.0000
Iteration  8192
 79494.0000 79536.0000    42.0000
Iteration  16384
 79585.0000 79670.0000    85.0000
***** INCOMPLETE MEASUREMENT *****

Test Name:   P000001                            Class Name:  Procedure
CPU Time:         7.9  microseconds
Wall Time:        7.9  microseconds.            Iteration Count:  16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters
```

```
Iteration  1
 79770.0000 79770.0000      0.0000
Iteration  2
 79770.0000 79770.0000      0.0000
Iteration  4
 79770.0000 79771.0000      1.0000
Iteration  8
 79771.0000 79771.0000      0.0000
Iteration  16
 79771.0000 79771.0000      0.0000
Iteration  32
 79772.0000 79772.0000      0.0000
Iteration  64
 79773.0000 79773.0000      0.0000
Iteration  128
 79774.0000 79775.0000      1.0000
Iteration  256
 79776.0000 79778.0000      2.0000
Iteration  512
 79780.0000 79783.0000      3.0000
Iteration  1024
 79788.0000 79794.0000      6.0000
Iteration  2048
 79803.0000 79816.0000     13.0000
Iteration  4096
 79833.0000 79860.0000     27.0000
Iteration  8192
 79894.0000 79947.0000     53.0000
Iteration  16384
 80014.0000 80120.0000    106.0000

Test Name:    P000010                     Class Name:  Procedure
CPU Time:        17.7  microseconds
Wall Time:       17.7  microseconds.      Iteration Count:   16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
 80257.0000 80257.0000      0.0000
Iteration  2
 80258.0000 80258.0000      0.0000
Iteration  4
 80258.0000 80258.0000      0.0000
Iteration  8
 80258.0000 80258.0000      0.0000
Iteration  16
 80259.0000 80259.0000      0.0000
Iteration  32
 80260.0000 80260.0000      0.0000
Iteration  64
 80262.0000 80262.0000      0.0000
Iteration  128
 80266.0000 80267.0000      1.0000
Iteration  256
 80273.0000 80275.0000      2.0000
Iteration  512
 80288.0000 80290.0000      2.0000
Iteration  1024
 80317.0000 80322.0000      5.0000
Iteration  2048
 80374.0000 80385.0000     11.0000

Test Name:    T000001                     Class Name:  Tasking
CPU Time:       449.2  microseconds
Wall Time:      449.2  microseconds.      Iteration Count:   2048
Test Description:
 Minimum rendezvous, entry call and return time
 1 task 1 entry , task inside procedure
 no select


Iteration  1
 80491.0000 80491.0000      0.0000
Iteration  2
 80491.0000 80491.0000      0.0000
Iteration  4
 80491.0000 80491.0000      0.0000
Iteration  8
 80492.0000 80492.0000      0.0000
Iteration  16
 80494.0000 80494.0000      0.0000
Iteration  32
 80496.0000 80496.0000      0.0000
Iteration  64
 80500.0000 80501.0000      1.0000
Iteration  128
 80508.0000 80509.0000      1.0000
Iteration  256
 80524.0000 80525.0000      1.0000
Iteration  512
 80555.0000 80558.0000      3.0000
Iteration  1024
 80618.0000 80623.0000      5.0000

Test Name:    T000004                     Class Name:  Tasking
CPU Time:       561.5  microseconds
Wall Time:      561.5  microseconds.      Iteration Count:   1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement


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

 Number of sample values is          7000
 Clock Resolution            =       1.000000000000000 seconds.
 Clock Resolution (average)  =       1.000000000000000 seconds.
 Clock Resolution (variance) =       0.000000000000000 seconds.


Test Name:    A000091             Class Name:  Composite
         1.1000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark


Test Name:    A000092                     Class Name: composite

Average time per cycle :  2824.67 milliseconds

Average Whetstone rating :  354 KWIPS
```

```
Iteration  1
 82935.0000 82935.0000      0.0000
Iteration  2
 82936.0000 82936.0000      0.0000
Iteration  4
 82936.0000 82936.0000      0.0000
Iteration  8
 82937.0000 82937.0000      0.0000
Iteration  16
 82938.0000 82938.0000      0.0000
Iteration  32
 82940.0000 82940.0000      0.0000
Iteration  64
 82944.0000 82944.0000      0.0000
Iteration  128
 82951.0000 82952.0000      1.0000
Iteration  256
 82965.0000 82967.0000      2.0000
Iteration  512
 82994.0000 82996.0000      2.0000
Iteration  1024
 83051.0000 83056.0000      5.0000

Test Name:    C000001                     Class Name:  Tasking
CPU Time:       996.1  microseconds
Wall Time:      996.1  microseconds.      Iteration Count:   1024
Test Description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop,

Iteration  1
 83167.0000 83167.0000      0.0000
Iteration  2
 83167.0000 83167.0000      0.0000
Iteration  4
 83168.0000 83168.0000      0.0000
Iteration  8
 83168.0000 83168.0000      0.0000
Iteration  16
 83169.0000 83170.0000      1.0000
Iteration  32
 83172.0000 83172.0000      0.0000
Iteration  64
 83176.0000 83176.0000      0.0000
Iteration  128
 83184.0000 83185.0000      1.0000
Iteration  256
 83199.0000 83200.0000      1.0000
Iteration  512
 83229.0000 83232.0000      3.0000
Iteration  1024
 83289.0000 83294.0000      5.0000

Test Name:    C000002                     Class Name:  Tasking
CPU Time:      1064.5  microseconds
Wall Time:     1064.5  microseconds.      Iteration Count:   1024
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop


Iteration  1
 83774.0000 83774.0000      0.0000
Iteration  2
 83774.0000 83774.0000      0.0000
Iteration  4
 83775.0000 83775.0000      0.0000
Iteration  8
 83775.0000 83775.0000      0.0000
Iteration  16
 83776.0000 83776.0000      0.0000
Iteration  32
 83777.0000 83778.0000      1.0000
Iteration  64
 83779.0000 83780.0000      1.0000
Iteration  128
 83783.0000 83784.0000      1.0000
Iteration  256
 83791.0000 83792.0000      1.0000
Iteration  512
 83805.0000 83808.0000      3.0000
Iteration  1024
 83834.0000 83840.0000      6.0000
Iteration  2048
 83892.0000 83904.0000     12.0000

Test Name:    H000004                     Class Name:  Chapter 13
CPU Time:       449.2  microseconds
Wall Time:      449.2  microseconds.      Iteration Count:   2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
 For this test the arrays are NOT PACKED with the pragma 'PACK.'
 For this test the operations are performed on components in a loop.

Iteration  1
 84523.0000 84523.0000      0.0000
Iteration  2
 84523.0000 84523.0000      0.0000
Iteration  4
 84523.0000 84523.0000      0.0000
Iteration  8
 84524.0000 84524.0000      0.0000
Iteration  16
 84524.0000 84524.0000      0.0000
Iteration  32
 84524.0000 84524.0000      0.0000
Iteration  64
 84525.0000 84525.0000      0.0000
Iteration  128
 84526.0000 84526.0000      0.0000
Iteration  256
 84527.0000 84529.0000      2.0000
Iteration  512
 84531.0000 84533.0000      2.0000
Iteration  1024
 84536.0000 84542.0000      6.0000
Iteration  2048
 84548.0000 84559.0000     11.0000
Iteration  4096
 84571.0000 84592.0000     21.0000
Iteration  8192
 84617.0000 84659.0000     42.0000
Iteration  16384
 84708.0000 84793.0000     85.0000
***** INCOMPLETE MEASUREMENT *****

Test Name:    P000001                     Class Name:  Procedure
CPU Time:         7.9  microseconds
```

```
Wall Time:       7.9  microseconds.          Iteration Count:   16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters


Iteration  1
 84893.0000 84893.0000       0.0000
Iteration  2
 84893.0000 84893.0000       0.0000
Iteration  4
 84893.0000 84893.0000       0.0000
Iteration  8
 84893.0000 84894.0000       1.0000
Iteration  16
 84894.0000 84894.0000       0.0000
Iteration  32
 84894.0000 84894.0000       0.0000
Iteration  64
 84895.0000 84896.0000       1.0000
Iteration  128
 84897.0000 84897.0000       0.0000
Iteration  256
 84899.0000 84900.0000       1.0000
Iteration  512
 84903.0000 84906.0000       3.0000
Iteration  1024
 84911.0000 84917.0000       6.0000
Iteration  2048
 84926.0000 84939.0000      13.0000
Iteration  4096
 84956.0000 84983.0000      27.0000
Iteration  8192
 85016.0000 85069.0000      53.0000
Iteration  16384
 85137.0000 85243.0000     106.0000

Test Name:   P000010                       Class Name:  Procedure
CPU Time:         17.7  microseconds
Wall Time:        17.7  microseconds.      Iteration Count:   16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
 85380.0000 85380.0000       0.0000
Iteration  2
 85380.0000 85380.0000       0.0000
Iteration  4
 85381.0000 85381.0000       0.0000
Iteration  8
 85381.0000 85381.0000       0.0000
Iteration  16
 85382.0000 85382.0000       0.0000
Iteration  32
 85383.0000 85383.0000       0.0000
Iteration  64
 85385.0000 85385.0000       0.0000
Iteration  128
 85389.0000 85389.0000       0.0000
Iteration  256
 85396.0000 85397.0000       1.0000
Iteration  512
 85411.0000 85413.0000       2.0000
Iteration  1024
 85440.0000 85445.0000       5.0000
Iteration  2048
 85497.0000 85508.0000      11.0000

Test Name:   T000001                       Class Name:  Tasking
CPU Time:       454.1  microseconds
Wall Time:      454.1  microseconds.       Iteration Count:   2048
Test Description:
 Minimum rendezvous, entry call and return time
 1 task 1 entry , task inside procedure
 no select


Iteration  1
 85614.0000 85614.0000       0.0000
Iteration  2
 85615.0000 85615.0000       0.0000
Iteration  4
 85615.0000 85615.0000       0.0000
Iteration  8
 85616.0000 85616.0000       0.0000
Iteration  16
 85617.0000 85617.0000       0.0000
Iteration  32
 85620.0000 85620.0000       0.0000
Iteration  64
 85624.0000 85624.0000       0.0000
Iteration  128
 85632.0000 85633.0000       1.0000
Iteration  256
 85648.0000 85649.0000       1.0000
Iteration  512
 85679.0000 85682.0000       3.0000
Iteration  1024
 85741.0000 85746.0000       5.0000

Test Name:   T000004                       Class Name:  Tasking
CPU Time:       556.6  microseconds
Wall Time:      556.6  microseconds.       Iteration Count:   1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement


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

 Number of sample values is          7000
 Clock Resolution            =        1.000000000000000 seconds.
 Clock Resolution (average)  =        1.000000000000000 seconds.
 Clock Resolution (variance) =        0.000000000000000 seconds.


Test Name:   A000091              Class Name:  Composite
          1.2000 is time in milliseconds for one Dhrystone
Test Description:
 Reinhold P. Weicker's DHRYSTONE composite benchmark


Test Name:   A000092                       Class Name: composite
```

```
Average time per cycle : -342758.67 milliseconds

Average Whetstone rating :  281 KWIPS

Iteration  1
   142.0000   142.0000       0.0000
Iteration  2
   142.0000   142.0000       0.0000
Iteration  4
   143.0000   143.0000       0.0000
Iteration  8
   144.0000   144.0000       0.0000
Iteration  16
   145.0000   145.0000       0.0000
Iteration  32
   147.0000   147.0000       0.0000
Iteration  64
   150.0000   151.0000       1.0000
Iteration  128
   158.0000   158.0000       0.0000
Iteration  256
   172.0000   173.0000       1.0000
Iteration  512
   201.0000   203.0000       2.0000
Iteration  1024
   257.0000   263.0000       6.0000

Test Name:   C000001                           Class Name:  Tasking
CPU Time:       996.1  microseconds
Wall Time:      996.1  microseconds.           Iteration Count:   1024
Test Description:
 Task create and terminate measurement
 with one task, no entries, when task is in a procedure
 using a task type in a package, no select statement, no loop,


Iteration  1
   373.0000   373.0000       0.0000
Iteration  2
   373.0000   373.0000       0.0000
Iteration  4
   374.0000   374.0000       0.0000
Iteration  8
   374.0000   374.0000       0.0000
Iteration  16
   376.0000   376.0000       0.0000
Iteration  32
   378.0000   378.0000       0.0000
Iteration  64
   381.0000   382.0000       1.0000
Iteration  128
   389.0000   390.0000       1.0000
Iteration  256
   404.0000   405.0000       1.0000
Iteration  512
   434.0000   437.0000       3.0000
Iteration  1024
   493.0000   499.0000       6.0000

Test Name:   C000002                           Class Name:  Tasking
CPU Time:      1035.2  microseconds
Wall Time:     1035.2  microseconds.           Iteration Count:   1024
Test Description:
 Task create and terminate time measurement.
 with one task, no entries when task is in a procedure,
 task defined and used in procedure, no select statement, no loop


Iteration  1
   978.0000   978.0000       0.0000
Iteration  2
   978.0000   978.0000       0.0000
Iteration  4
   979.0000   979.0000       0.0000
Iteration  8
   979.0000   979.0000       0.0000
Iteration  16
   980.0000   980.0000       0.0000
Iteration  32
   981.0000   982.0000       1.0000
Iteration  64
   983.0000   984.0000       1.0000
Iteration  128
   987.0000   988.0000       1.0000
Iteration  256
   995.0000   996.0000       1.0000
Iteration  512
  1009.0000  1012.0000       3.0000
Iteration  1024
  1038.0000  1044.0000       6.0000
Iteration  2048
  1096.0000  1108.0000      12.0000

Test Name:   H000004                           Class Name:  Chapter 13
CPU Time:       444.3  microseconds
Wall Time:      444.3  microseconds.           Iteration Count:   2048
Test Description:
 Time to perform standard boolean operations on arrays of booleans.
 For this test the arrays are NOT PACKED with the pragma 'PACK.'
 For this test the operations are performed on components in a loop.


Iteration  1
  1798.0000  1798.0000       0.0000
Iteration  2
  1798.0000  1798.0000       0.0000
Iteration  4
  1798.0000  1798.0000       0.0000
Iteration  8
  1798.0000  1798.0000       0.0000
Iteration  16
  1799.0000  1799.0000       0.0000
Iteration  32
  1799.0000  1799.0000       0.0000
Iteration  64
  1800.0000  1800.0000       0.0000
Iteration  128
  1801.0000  1802.0000       1.0000
Iteration  256
  1803.0000  1804.0000       1.0000
Iteration  512
  1806.0000  1809.0000       3.0000
Iteration  1024
  1812.0000  1818.0000       6.0000
Iteration  2048
  1825.0000  1837.0000      12.0000
Iteration  4096
  1850.0000  1874.0000      24.0000
Iteration  8192
  1899.0000  1947.0000      48.0000
```

```
Iteration  16384
  1998.0000  2093.0000    95.0000

Test Name:    P000001                       Class Name:  Procedure
CPU Time:          4.3  microseconds
Wall Time:         4.3  microseconds.       Iteration Count:   16384
Test Description:
 Procedure call and return time ( may be zero if automatic inlining )
 procedure is local
 no parameters


Iteration  1
  2197.0000  2197.0000     0.0000
Iteration  2
  2198.0000  2198.0000     0.0000
Iteration  4
  2198.0000  2198.0000     0.0000
Iteration  8
  2198.0000  2198.0000     0.0000
Iteration  16
  2198.0000  2199.0000     1.0000
Iteration  32
  2199.0000  2199.0000     0.0000
Iteration  64
  2200.0000  2200.0000     0.0000
Iteration  128
  2201.0000  2202.0000     1.0000
Iteration  256
  2203.0000  2204.0000     1.0000
Iteration  512
  2207.0000  2210.0000     3.0000
Iteration  1024
  2214.0000  2221.0000     7.0000
Iteration  2048
  2229.0000  2242.0000    13.0000
Iteration  4096
  2259.0000  2285.0000    26.0000
Iteration  8192
  2318.0000  2370.0000    52.0000
Iteration  16384
  2436.0000  2539.0000   103.0000

Test Name:    P000010                       Class Name:  Procedure
CPU Time:         18.3  microseconds
Wall Time:        18.3  microseconds.       Iteration Count:   16384
Test Description:
 Procedure call and return time measurement
 Compare to P000005
 10 parameters, in INTEGER


Iteration  1
  2674.0000  2674.0000     0.0000
Iteration  2
  2675.0000  2675.0000     0.0000
Iteration  4
  2675.0000  2675.0000     0.0000
Iteration  8
  2675.0000  2675.0000     0.0000
Iteration  16
  2676.0000  2676.0000     0.0000
Iteration  32
  2677.0000  2677.0000     0.0000
Iteration  64
  2679.0000  2679.0000     0.0000
Iteration  128
  2683.0000  2684.0000     1.0000
Iteration  256
  2690.0000  2692.0000     2.0000
Iteration  512
  2705.0000  2708.0000     3.0000
Iteration  1024
  2734.0000  2739.0000     5.0000
Iteration  2048
  2791.0000  2802.0000    11.0000

Test Name:    T000001                       Class Name:  Tasking
CPU Time:        454.1  microseconds
Wall Time:       454.1  microseconds.       Iteration Count:   2048
Test Description:
 Minimum rendezvous, entry call and return time
 1 task 1 entry , task inside procedure
 no select


Iteration  1
  2908.0000  2908.0000     0.0000
Iteration  2
  2908.0000  2908.0000     0.0000
Iteration  4
  2909.0000  2909.0000     0.0000
Iteration  8
  2910.0000  2910.0000     0.0000
Iteration  16
  2911.0000  2911.0000     0.0000
Iteration  32
  2913.0000  2913.0000     0.0000
Iteration  64
  2917.0000  2918.0000     1.0000
Iteration  128
  2925.0000  2926.0000     1.0000
Iteration  256
  2942.0000  2943.0000     1.0000
Iteration  512
  2974.0000  2977.0000     3.0000
Iteration  1024
  3038.0000  3044.0000     6.0000

Test Name:    T000004                       Class Name:  Tasking
CPU Time:        566.4  microseconds
Wall Time:       566.4  microseconds.       Iteration Count:   1024
Test Description:
 Task entry call and return time measured
 One tasks active, two entries, tasks in a package
 using select statement
```

Bibliography

1. Boehm, B.W., *A Spiral Model of Software Development and Enhancement,* in *Software Engineering Project Management,* R.H. Thayer, Editor. 1987, IEEE Computer Society Press: Los Alamitos, CA. p. 128 - 142.

2. Yeh, R.T., *et al.*, *A Commonsense Management Model,* in *IEEE Software*. 1991, p. 23 - 33.

3. Pyster, A., *The Synthesis Process for Software Development,* in *System and Software Requirements Engineering,* R.H. Thayer and M. Dorfman, Editor. 1990, IEEE Computer Society Press: Los Alamitos, CA. p. 528 - 538.

4. Shaw, A.C., *Reasoning About Time in Higher-Level Language Software.* IEEE Transactions on Software Engineering, 1989. **15**(7): p. 875 - 889.

5. Park, C.Y. and A.C. Shaw. *A Source-Level Tool for Predicting Deterministic Execution Times of Programs*. Department of Computer Science, University of Washington, (Technical Report 89-09-12). September 13, 1989.

6. Shaw, A.C. *Towards a Timing Semantics For Programming Languages.* in *Third Annual Workshop, Foundations of Real-Time Computing*. 1990. Washington, DC: Office of Naval Research.

7. Goos, G., W.A. Wulf, A. Evans Jr., and K.J. Butler, ed. *DIANA: An Intermediate Language for Ada*. Lecture Notes in Computer Science, ed. G. Goos and J. Hartmanis. 1983, Springer-Verlag: Berlin. 201 pages.

8. Shaw, A.C. *Communiating Real-Time State Machines*. Department of Computer Science and Engineering, University of Washington, (Technical Report 91-08-09). August 1991.

9. Haase, V.H., *Real-Time Behavior of Programs.* IEEE Transactions on Software Engineering, 1981. **7**(5): p. 494 - 501.

10. Halang, W.A. *A Priori Execution Time Analysis for Parallel Processes*. in *Proceedings of the Euromicro Workshop on Real-Time*. 1989. IEEE Computer Society Press.

11. Mok, A.K., P. Amerasinghe, M. Chen, and K. Tantisirivat. *Evaluating Tight Execution Time Bounds of Programs by Annotations*. in *6th IEEE Workshop on Real-Time Operating Systems and Software*. 1989. Pittsburgh:

12. Puschner, P. and C. Koza, *Calculating the Maximum Execution Time of Real-Time Programs*. The Journal of Real-Time Systems, 1989. **1**(2): p. 159 - 176.

13. Kenny, K.B. and K.-J. Lin, *Measuring and Analyzing Real-Time Performance,* in *IEEE Software*. 1991, p. 41 - 49.

14. Glicker, S.M. and F.A. Hosch. *Toward Automating the Execution Timing Analysis of Ada Tasks in Event-Driven Real-Time Systems*. Applied Research Laboratories, The University of Texas at Austin, (Technical Report ARL-TR-91-4). 19 February 1991.

15. Knuth, D.E., *The Art of Computer Programming*. Second ed. Addison-Wesley Series in Computer Science and Information Processing, ed. R.S. Varga and M.A. Harrison. Vol. 1/Fundamental Algorithms. 1973, Reading, Massachusetts: Addison-Wesley Publishing Company. 634 pages.

16. Hoare, C.A.R., *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 1969. **12**(10): p. 576-580.

17. Shaw, M. *A formal System for Specifying and Verifying Program Performance*. Department of Computer Science, Carnegie-Mellon University, (Technical Report CMU-CS-79-129). 21 June 1979.

18. Walden, E. and C.V. Ravishankar, *A Survey of Hard Real-Time Scheduling Algorithms*. unpublished draft, 1990. .

19. Cornhill, D., *et al*. *Limitations of Ada for Real-Time Scheduling*. in *Proceedings of the International Workshop of Real-Time Ada Issues*. 1987. Moretonhampstead, Devon, UK: ACM SIGAda.

20. Baker, T.P. and A. Shaw. *The Cyclic Executive Model and Ada*. in *The Real-Time Systems Symposium*. 1988. Huntsville, AL: IEEE Computer Society.

21. Grogono, P., *Comments, Assertions, and Pragmas*. SIGPLAN Notices, 1989. **24**(3): p. 79 - 84.

22. Lin, K.-J. and J.W.S. Liu. *FLEX: A Language for Real-Time Systems Programming*. in *Third Annual Workshop, Foundations of Real-Time Computing*. 1990. Washington, DC: Office of Naval Research.

23. *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, Ada Joint Program Office, (January 1983).

24. Motorola, *MC68030 Enhanced 32-Bit Microprocessor User's Manual*. Third ed. 1990, Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

25. Shin, K.G. *HARTS: A Distributed Real-Time Architecture*. in *Third Annual Workshop, Foundations of Real-Time Computing*. 1990. Washington, DC: Office of Naval Research.

26. Stankovic, J.A. and K. Ramamritham, *The Spring Kernel: A New Paradigm for Real-Time Operating Systems*. SIGOPS, 1989. **23**(3): p. 54 - 71.

27. Chen, M.-S., K.G. Shin, and D.D. Kandlur, *Addressing, Routing, and Broadcasting in Hexagonal Mesh Multiprocessors*. IEEE Transactions on Computers 1990. **39**(1): p. 10 - 18.

28. Hoare, C.A.R., *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, ed. C.A.R. Hoare. 1985, Englewood Cliffs, New Jersey: Prentice-Hall. 256 pages.

29. Hoare, C.A.R., *Communicating Sequential Processes*. Communications of the ACM, 1978. **21**(8): p. 666-677.

30. Park, C.Y. and A.C. Shaw. *Experiments With a Program Timing Tool Based on Source-Level Timing Schema*. in *IEEE Real-Time Systems Symposium*. 1990. Lake Buena Vista, Florida: IEEE Computer Society Press.

31. Woodbury, M.H., *Workload Characterization of Real-Time Computing Systems*. 1988, The University of Michigan:

32. Chu, W.W., C.-M. Sit, and K.K. Leung, *Task Response Time For Real-Time Distributed Systems With Resource Contentions*. IEEE Transactions on Software Engineering, 1991. **17**(10): p. 1076 - 1092.

33. Gerber, R. and I. Lee. *Communicating Shared Resources: A Model for Distributed Real-Time Systems*. in *IEEE Real-Time Systems Symposium*. 1989. Santa Monica, CA: IEEE Computer Society Press.