**Execution Time Prediction of Ada Programs**

**Christopher A. Warack, Captain**

**AFIT Student Attending: University of Michigan**

**Approved for Public Release IAW 190-1**
Distributed Unlimited

**ERNEST A. HAYGOOD, Captain, USAF**
Executive Officer

**Abstract**

**AFIT/CI/CIA-91-134**

**Report Number**

**MAY 7 1992**

**Wright-Patterson AFB OH 45433-6583**

**Page Dimensions:** 614.4 x 791.0

**Form Approved**

**OMB No 0704-0188**

**Washington Headquarters Services, Executive Office for Information Operations and Reports**

**Time and Effort Reduction Project (DFSS 0188) Washington, DC 20525**

**Distributed Unlimited**

**Security Classification of Report:**

**Security Classification of This Page:**

**Security Classification of Abstract:**

**Limitation of Abstract:**

**Number of Pages:** 121
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 Ilsi programming environment. A foundation is laid for measuring how good a timing analysis prediction fits the implementation.
Bibliography


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 IIIsi 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

Abstract ............................................................................................................... ii
Acknowledgement ............................................................................................ iii
Introduction ........................................................................................................ 1
  Motivation ........................................................................................................ 1
  Objectives ......................................................................................................... 3
  Organization ..................................................................................................... 4
Execution Time Prediction and Scheduling ....................................................... 6
  Introduction ..................................................................................................... 6
  Traditional Execution Time Prediction Methods ........................................... 7
  Schema Based Timing Analysis ..................................................................... 10
  Scheduling Analysis ....................................................................................... 13
  Discussion ....................................................................................................... 15
Timing Schema and Events ................................................................................ 16
  Introduction ..................................................................................................... 16
  DIANA Representation of Ada Programs ...................................................... 16
  Timing Analysis Transformation Algorithm ................................................ 20
  Timing Schema for DIANA Objects ............................................................... 21
  Event Structure for DIANA Objects ............................................................... 28
  Assertions ...................................................................................................... 30
  Compiler Analysis .......................................................................................... 33
  Execution Time Prediction Algorithm ........................................................... 35
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 \( x \in \{ \text{all possible execution conditions} \} \). 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 \( T = [t_{\text{min}}, t_{\text{max}}] \) 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, \( T \circ x = [t_{\text{min}} \circ x, t_{\text{max}} \circ x] \). Relational operations are applied to the \( t_{\text{max}} \) component first since the worst-
case is the more important in real-time analysis. If the \( t_{\text{max}} \) components are equal, then comparison of the \( t_{\text{min}} \) components determines the result.

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

\[
T[\text{if-statement}] = [10,15] + T[\text{boolean-exp}] + [\min((4,6) + T[\text{then-part}]), (2,4) + T[\text{else-part}])]
\]

where if-statement ::= if boolean-exp then then-part else else-part;

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 \( \langle a, c, d \rangle \) 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 analyser 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

\[ \text{name}^1 : \text{node_type} \{ \text{attributes} \} \]. Attributes are juxtaposed pairs of the form \( \text{attribute_name} \text{ attribute_value} \). Multiple attributes are separated by semicolons.

```ada
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].
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_id ^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 ^PD9;
  sm_defn ^A24 ]

A39 : function_call  [ as_name ^A40;
  as_param_assoc_s ^A41 ]

A40 : used_btn_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_btn_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.
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(DSCTRLRANGE):** Determines the lowest and highest possible values of the range.
<table>
<thead>
<tr>
<th>Node Type</th>
<th>Arity</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>1</td>
<td>Abort -- abort statements are treated as errors at this point and are not candidates for a min or max path.</td>
</tr>
<tr>
<td>accept</td>
<td>3</td>
<td>P(accept) + Node(&quot;start &amp; as name&quot;) + Node(&quot;Begin&amp;as name&quot;) + P(rendezvous) + T(as stm s) + P(acceptend) + Node(&quot;end &amp; as name&quot;)</td>
</tr>
<tr>
<td>access</td>
<td>1</td>
<td>T(as_constrained)</td>
</tr>
<tr>
<td>address</td>
<td>2</td>
<td>[0,0] -- representation clause affects compilation only</td>
</tr>
<tr>
<td>aggregate</td>
<td>n</td>
<td>[\sum_{i=1}^{n} T(as_list[i]) + Store(sm_exp_type)]</td>
</tr>
<tr>
<td>alignment</td>
<td>2</td>
<td>[0,0] -- representation clause affects compilation only</td>
</tr>
<tr>
<td>all</td>
<td>1</td>
<td>[0,0];</td>
</tr>
<tr>
<td>allocator</td>
<td>1</td>
<td>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)))</td>
</tr>
<tr>
<td>alternative</td>
<td>2</td>
<td>-- subsumed in schema for case node</td>
</tr>
<tr>
<td>alternative_s</td>
<td>n</td>
<td>-- subsumed in schema for case node</td>
</tr>
<tr>
<td>and _then</td>
<td>0</td>
<td>P(and _then)</td>
</tr>
<tr>
<td>argument_id</td>
<td>0</td>
<td>[0,0] -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>array</td>
<td>2</td>
<td>T(as descr_range ..a) + T(as constrained)</td>
</tr>
<tr>
<td>assign</td>
<td>2</td>
<td>T(as_name) + T(as_exp) + Store(as_name) + ConstraintCheck(as_name) + if as_exp = used_object then Access(as_exp.sm_exp_type) else [0,0]</td>
</tr>
<tr>
<td>assoc</td>
<td>2</td>
<td>T(as_actual)</td>
</tr>
<tr>
<td>attr_id</td>
<td>0</td>
<td>[0,0] -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>attribute</td>
<td>2</td>
<td>T(as_name) + P(&quot;attr_&quot; &amp; as_id.sm_defn.lx_symrep) -- attribute execution times are pre-defined</td>
</tr>
<tr>
<td>attribute_call</td>
<td>2</td>
<td>T(as_exp) + T(as_name) -- as_name is a node of type attribute</td>
</tr>
<tr>
<td>binary</td>
<td>3</td>
<td>T(as_exp1) + T(as_exp2) + T(as_binary_op)</td>
</tr>
<tr>
<td>block</td>
<td>3</td>
<td>T(as.item_s) + T(as.stm_s) -- as_alternative_s represents exception handlers</td>
</tr>
<tr>
<td>box</td>
<td>0</td>
<td>[0,0] -- generic subprogram formal option</td>
</tr>
<tr>
<td>case</td>
<td>2</td>
<td>T(as_exp) + P(case) + [min(choices),max(choices)] choices = T(as_alternative_s.as_list[n].as stm s) (1 \leq n \leq \mid as_list) -- as_alternative_s.as_list[n].as_choice_s must be static and is ignored</td>
</tr>
<tr>
<td>choice_s</td>
<td>n</td>
<td>[\sum_{i=1}^{n} T(as_list[i])]</td>
</tr>
<tr>
<td>code</td>
<td>2</td>
<td>T(as_exp) -- machine dependent code insertion -- execution time bounds must be specified with a time assertion</td>
</tr>
<tr>
<td>comp_id</td>
<td>0</td>
<td>[0,0] -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>comp_rep</td>
<td>3</td>
<td>[0,0] -- representation clause affects compilation only</td>
</tr>
<tr>
<td>comp_rep_s</td>
<td>n</td>
<td>[0,0] -- representation clause affects compilation only</td>
</tr>
<tr>
<td>comp_unit</td>
<td>3</td>
<td>T(as_unit.body) -- as_context and as pragma_s set up environment only</td>
</tr>
<tr>
<td>compilation</td>
<td>n</td>
<td>(map T as_list = T(hd(as_list))(map T tl(as_list))) -- list of all compilation units in the system</td>
</tr>
<tr>
<td>cond_clause</td>
<td>2</td>
<td>-- subsumed in schema for &quot;if&quot; node</td>
</tr>
<tr>
<td>cond_entry</td>
<td>2</td>
<td>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</td>
</tr>
<tr>
<td>const_id</td>
<td>0</td>
<td>[0,0] -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>constant</td>
<td>3</td>
<td>T(as_type_spec) + T(as_object_def)</td>
</tr>
<tr>
<td>constrained</td>
<td>2</td>
<td>T(as_constrained)</td>
</tr>
<tr>
<td>context</td>
<td>n</td>
<td>[0,0] -- set's up environment, no cost</td>
</tr>
</tbody>
</table>
conversion  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(am_exp_type) else (0,0)
    + T(as_exp)

decl_s n 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)ndef_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)dscrnt Aggregate n n \sum_{i=1}^{n} T(as_list[i])dscrnt_id 0 [0,0] -- identifier "symbol table" entrydscrnt_var 3 T(as_object_def)dscrnt_var_s n n \sum_{i=1}^{n} T(as_list[i])dscrnt_range_s n n \sum_{i=1}^{n} T(as_list[i])entry 2 T(as_dscrnt_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" entryenum_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
  each elaboration timeexception 2 [0,0] -- declares exception namesexception_id 0 [0,0] -- identifier "symbol table" entryexit 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
  \footnote{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.}exp_s n 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_dscrnt_range)formal_dscrnt 0 [0,0] -- place holder for generic type parametersformal_fixed 0 [0,0] -- place holder for generic type parametersformal_float 0 [0,0] -- place holder for generic type parametersformal_integer 0 [0,0] -- place holder for generic type parametersfunction 2 T(as_param_s)
25

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 n
  \sum_{i=1}^{n} T(as_list(i))

generic_id 0 (0,0) -- identifier "symbol table" entry

generic_param_s n 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 choices = \{ \left( \sum_{i=1}^{n} T(as_list(i)), as_exp_void \right) +

  T(as_list(n), as_stm_s) : 1\leq|as_list| \}
  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 unconstrained type.

indexed 2 T(as_name) + T(as_exp_s);

inner_record n n
  \sum_{i=1}^{n} T(as_list(i))

instantiation 2 T(as_generic_assoc_s)

type 0 T(as_range),

item_s n 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_dacrt_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)
<table>
<thead>
<tr>
<th>name_s</th>
<th>n</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>named</td>
<td>2</td>
<td>( T(\text{as_choice_s}) + T(\text{as_exp}) )</td>
</tr>
<tr>
<td>named_stm</td>
<td>2</td>
<td>( T(\text{as_stm}) )</td>
</tr>
<tr>
<td>named_stm_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>no_default</td>
<td>0</td>
<td>([0,0]) -- generic subprogram formal option</td>
</tr>
<tr>
<td>not_in</td>
<td>0</td>
<td>( P(\text{not_in}) )</td>
</tr>
<tr>
<td>null_access</td>
<td>0</td>
<td>([0,0])</td>
</tr>
<tr>
<td>null_comp</td>
<td>0</td>
<td>([0,0])</td>
</tr>
<tr>
<td>null_stm</td>
<td>0</td>
<td>( P(\text{null}) )</td>
</tr>
<tr>
<td>number</td>
<td>2</td>
<td>([0,0]) -- static numeric constant</td>
</tr>
<tr>
<td>number_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>numeric_literal</td>
<td>0</td>
<td>( \text{if sm_value} &lt; \text{SMALL_VAL} ) then ( P(\text{small_num_literal_access}) ) else ( P(\text{num_literal_access}) ) -- where num is int, float or fixed</td>
</tr>
<tr>
<td>or_else</td>
<td>0</td>
<td>( P(\text{or_else}) )</td>
</tr>
<tr>
<td>others</td>
<td>0</td>
<td>([0,0])</td>
</tr>
<tr>
<td>out</td>
<td>3</td>
<td>([0,0]) -- cannot have default parameters</td>
</tr>
<tr>
<td>out_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>package_body</td>
<td>2</td>
<td>( \text{save id(\text{as_id} &amp; &quot;body&quot;)} = P(\text{package_elaboration}) + T(\text{as_block_stub}) )</td>
</tr>
<tr>
<td>package_decl</td>
<td>2</td>
<td>( \text{save id(\text{as_id})} = T(\text{as_package_def}) + P(\text{package_elaboration}) ) -- if as_package_def is rename or instantiation then use other values</td>
</tr>
<tr>
<td>package_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>package_spec</td>
<td>2</td>
<td>( T(\text{as_decl_s1}) + T(\text{as_decl_s2}) )</td>
</tr>
<tr>
<td>param_assoc_s</td>
<td>n</td>
<td>( \sum_{i=1}^{n} T(\text{as_list}(i)) ) -- actual parameter lists</td>
</tr>
<tr>
<td>param_s</td>
<td>n</td>
<td>( \sum_{i=1}^{n} T(\text{as_list}(i)) )</td>
</tr>
<tr>
<td>parenthesized</td>
<td>1</td>
<td>( T(\text{as_exp}) )</td>
</tr>
<tr>
<td>pragma</td>
<td>2</td>
<td>([0,0]) -- compiler directive may change global params, but no code gen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing Assertion pragmas are handled, of course.</td>
</tr>
<tr>
<td>pragma_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>pragma_s</td>
<td>n</td>
<td>([0,0]) -- compiler directive may change global params, but no code gen</td>
</tr>
<tr>
<td>private</td>
<td>0</td>
<td>([0,0])</td>
</tr>
<tr>
<td>private_type_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>proc_id</td>
<td>0</td>
<td>([0,0]) -- identifier &quot;symbol table&quot; entry</td>
</tr>
<tr>
<td>procedure</td>
<td>1</td>
<td>( T(\text{as_param_s}) )</td>
</tr>
<tr>
<td>procedure_call</td>
<td>2</td>
<td>( T(\text{sm_normalized_param_s}) + P(\text{procedure_call}) + \text{Insert(as_name)} + P(\text{procedure_end}) ) -- sm_normalized_param_s includes default params</td>
</tr>
<tr>
<td>qualified</td>
<td>2</td>
<td>( T(\text{as_exp}) ) -- as_name only used by compiler</td>
</tr>
<tr>
<td>raise</td>
<td>1</td>
<td>Abort -- exceptions are treated as errors at this point and are not candidates for a min or max path.</td>
</tr>
<tr>
<td>range</td>
<td>2</td>
<td>( T(\text{as_exp1}) + T(\text{as_exp2}) )</td>
</tr>
<tr>
<td>record</td>
<td>n</td>
<td>( \sum_{i=1}^{n} T(\text{as_list}(i)) )</td>
</tr>
<tr>
<td>record_rep</td>
<td>3</td>
<td>([0,0]) -- representation clause affects compilation only</td>
</tr>
<tr>
<td>rename</td>
<td>1</td>
<td>([0,0])</td>
</tr>
<tr>
<td>return</td>
<td>1</td>
<td>( T(\text{as_exp_void}) + \text{Store(function_call)} + \text{Stop} )</td>
</tr>
<tr>
<td>reverse</td>
<td>2</td>
<td>( T(\text{as_dscrt_range}) )</td>
</tr>
<tr>
<td>select</td>
<td>2</td>
<td>( P(\text{select}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{as_select_clauses_s} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( + \sum_{i=1}^{\text{as_select_clauses_s}} T(\text{as_list}(i).\text{as_exp_void}) ) -- comp guards</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( + {\text{map (Xx.Nodel + Tx.as_stm_s) + Node2} \text{, as_select_clauses_s.\text{as_list} }) and if (</td>
</tr>
<tr>
<td>select_clause</td>
<td>2</td>
<td>-- subsumed in schema for select node</td>
</tr>
<tr>
<td>select_clause_s</td>
<td>n</td>
<td>-- subsumed in schema for select node</td>
</tr>
<tr>
<td>-----------------</td>
<td>---</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>selected</td>
<td>2</td>
<td>if as_name.sm_obj_type is a record then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if it has a variant then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T(as_name) + P(variant_tag_check) + T(as_designator_char)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T(as_name) + T(as_designator_char);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else -- it is an expanded name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T(as_name) + T(as_designator_char);</td>
</tr>
</tbody>
</table>

| simple_rep      | 2 | [0,0] -- representation clause affects compilation only |
| slice           | 2 | T(as_name) + T(as_dscre_range); |
|                 |   | size = as_dscre_range * sm_exp_type.as_constrained.cd_impl_size |

| stm_s           | n | 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 | 1 | 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_dscrent_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 | 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 \( \text{sm}_\text{value} \) 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 \( \text{sm}_\text{value} \) 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.

```
Task_Body

Enclosing Block

Accept

Entry_Call

Delay

Cond_Entry

Timed_Entry

Select
```

where `**` = `T(accept), T(delay), T(terminate)` or, for branch $n$, `delay(r)` = else branch.
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.

![Diagram of stitching together an accept statement and entry call](image)

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.
<table>
<thead>
<tr>
<th>Assertion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TA_Loop_Bound</strong></td>
<td>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.</td>
</tr>
<tr>
<td>(Low, High : natural)</td>
<td></td>
</tr>
<tr>
<td><strong>TA_Recursion_Bound</strong></td>
<td>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.</td>
</tr>
<tr>
<td>(Low, High : natural)</td>
<td></td>
</tr>
<tr>
<td><strong>TA_Measure_Start</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>TA_Measure_Stop</strong></td>
<td>Ignore following code in this compilation unit for timing analysis purposes. This assertion is intended to mark where the end event should be inserted.</td>
</tr>
<tr>
<td><strong>TA_Time_Absolute</strong></td>
<td>This assertion is treated exactly like a code sequence which reduces to an edge with time bounds [low, high] (in cycles).</td>
</tr>
<tr>
<td>(Low, High : Natural)</td>
<td></td>
</tr>
<tr>
<td><strong>TA_Time_Mix</strong></td>
<td>This assertion is similar to TA_Time_Absolute but uses an instruction count and average instruction time range (mix) to compute the time bounds.</td>
</tr>
<tr>
<td>(Instruction_Number : Natural; Mix : Mix_Type)</td>
<td></td>
</tr>
<tr>
<td><strong>TA_Time_By_Primitives</strong></td>
<td>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.</td>
</tr>
<tr>
<td>(&lt;prim_name&gt; =&gt; Natural, _)</td>
<td></td>
</tr>
</tbody>
</table>
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\textsuperscript{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

\footnote{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 runnable 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 runnable. 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. In the worst case, a task blocks in all of these situations and a context switch occurs. A task may become runnable 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.

```plaintext
procedure DOIT is
  COUNT_A, COUNT_B : INTEGER := 0;
  RESULT_C : INTEGER := 6;

  task TASK_A is
    COUNTB := COUNTB + COUNTB;
  end TASK_A;

  task TASK_B is
    COUNT_B := COUNT_B + I;
    TASK_B.CENTRY_B;
    COUNT_B := COUNT_B + COUNT_B;
  end TASK_B;

  task TASK_C is
    for I in 1 .. 2 loop
      entry ENTRY_A;
      entry ENTRY_B;
      entry DONE;
    end TASK_C;

  task body TASK_A is
    for I in 1 .. 100 loop
      COUNT_A := COUNT_A + 1;
    end loop;
    TASK_C.ENTRY_A;
    COUNT_A := COUNT_A + COUNT_A;
  end TASK_A;

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

  task TASK_A is
    COUNTB := COUNTB + COUNTB;
  end TASK_A;

  task TASK_B is
    for I in 1 .. 100 loop
      COUNT_B := COUNT_B + I;
      TASK_B.CENTRY_B;
      COUNT_B := COUNT_B + COUNT_B;
    end TASK_B;

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

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

Figure 6: Simple Tasking Program
Figure 7: Timing Graphs for Simple Tasking Program

<table>
<thead>
<tr>
<th>TASK_A</th>
<th>TASK_B</th>
<th>TASK_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(\text{for_loop}) + 1000 \cdot P(\text{integer_store}) + P(\text{int_plus}) + P(\text{int_literal_access}) + P(\text{integer_access}) + P(\text{for_iter}) - P(\text{for_iter}) + P(\text{for_end}) + P(\text{queue_entry}) )</td>
<td>( P(\text{integer_store}) + P(\text{int_plus}) + 2 \cdot P(\text{integer_access}) + P(\text{for_iter}) + P(\text{for_end}) + P(\text{queue_entry}) )</td>
<td>( P(\text{for_loop}) + P(\text{select}(2)) )</td>
</tr>
</tbody>
</table>

ENTRY_A  ENTRY_A  ENTRY_A

\( P(\text{activ}) \)

\( P(\text{for\_iter}) \)

ENTRY_B  ENTRY_B

\( P(\text{for\_end}) + P(\text{queue\_entry}) \)

DONE  DONE

\( P(\text{terminates}) \)

\( P(\text{for\_end}) + P(\text{accept}) \)

\( P(\text{terminates}) \)

\( P(\text{for\_end}) + P(\text{accept}) \)

** = \( P(\text{rendezvous}) + P(\text{integer\_store}) + P(\text{int\_divide}) + P(\text{int\_literal\_access}) + P(\text{access\_int\_var}) \)

\* = \( P(\text{rendezvous}) + P(\text{integer\_store}) + P(\text{int\_plus}) + P(\text{integer\_literal\_access}) + P(\text{access\_int\_var}) \)

\( 3 \cdot P(\text{integer\_access}) + P(\text{queue\_entry}) \) + \( 3 \cdot P(\text{task\_spec\_elab}) + 3 \cdot P(\text{task\_body\_elab}) + P(\text{activation}) \)

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 \(\rightarrow\) 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 \([\text{min}, \text{max}]\) 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 a time. A trace begins with the event of interest and an arbitrary runnable 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 runnable 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 and 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. 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:

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

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

$$T_{ex} = \frac{T_{measured} - (T_{clock} + T_{loop-overhead})}{I} \approx \frac{T_{measured}}{I}$$

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 $31.3 - 37.3\%$. 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.

<table>
<thead>
<tr>
<th>PIWG Experiment</th>
<th>Iterations</th>
<th>Predicted Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIWG Experiment</td>
<td>Iterations</td>
<td>Predicted Iterations</td>
</tr>
<tr>
<td>C000001 T</td>
<td>27 - 27</td>
<td>108 - 108</td>
</tr>
<tr>
<td>C000002 T</td>
<td>28 - 29</td>
<td>113 - 113</td>
</tr>
<tr>
<td>H000004 T</td>
<td>13 - 13</td>
<td>52 - 52</td>
</tr>
<tr>
<td>P000001 C</td>
<td>1 - 2</td>
<td>5 - 6</td>
</tr>
<tr>
<td>P000001 T</td>
<td>1 - 2</td>
<td>5 - 6</td>
</tr>
<tr>
<td>P000010 T</td>
<td>2 - 3</td>
<td>8 - 8</td>
</tr>
<tr>
<td>T000001 T</td>
<td>13 - 13</td>
<td>51 - 52</td>
</tr>
<tr>
<td>T000004 T</td>
<td>30 - 30</td>
<td>119 - 121</td>
</tr>
</tbody>
</table>

Table 1: Iterated Experiment Results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Measured Time</th>
<th>Predicted Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A000091</td>
<td>1.10 - 1.20 ms</td>
<td></td>
</tr>
<tr>
<td>A000092</td>
<td>2.78 - 2.84 ms</td>
<td></td>
</tr>
<tr>
<td>SimpleTasks</td>
<td>11.8 - 14.6</td>
<td></td>
</tr>
</tbody>
</table>

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.
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.
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 architecture 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 compiler's 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 modeled 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 IIIsi 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 IIIsi executes at 25 MHz.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>active</td>
<td>184</td>
<td>618</td>
</tr>
<tr>
<td>activation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocate_main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and_then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_aggr_comp_access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_aggr_setup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_cat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_comp_access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_comp_store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_ge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_greater</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_le</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_lesser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_not_in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_aft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_callable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_constrained</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_delta</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_digits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_emax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_epsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_first</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_first(N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_first_bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_fore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_last(N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_last_bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_length(N)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_emax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_emin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_mantissa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_overflows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_radix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_machine_rounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attr_mantissa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
case switch
case convert_array
case convert_derived
case convert_fixed2float
case convert_fixed2int
case convert_float2fixed
case convert_float2int
case convert_int2fixed
case convert_int2float
default_param
delay
delay else
else fixed_abs
else fixed_access
else fixed_div
else fixed_eq
else fixed_ge
else fixed_greater
else fixed_identity
else fixed_in
else fixed_le
else fixed_lesser
else fixed_minus
else fixed_mul
else fixed_neg
else fixed_negation
else fixed_not_in
else fixed_plus
else fixed_store
else float_access
else float_abs
else float_div
else float_eq
else float_exp
else float_identity
else float_in
else float_minus
else float_mul
else float_neg
else float_negation
else float_not_in
else float_plus
else float_store
else for_end
else for_iter
else for_loop
else function_access
else function_call
else function_end
else if
else int_abs
else int_div
else int_eq
else int_exp
else int_ge
else int_greater
else int_idenity
else int_in
else int_le
else int_lesser
else int_literal_access
else int_minus
else int_mod
else int_neg
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int_negation</td>
<td></td>
</tr>
<tr>
<td>int_not_in</td>
<td></td>
</tr>
<tr>
<td>int_plus</td>
<td>8 12</td>
</tr>
<tr>
<td>int_rem</td>
<td></td>
</tr>
<tr>
<td>int_times</td>
<td></td>
</tr>
<tr>
<td>integer_access</td>
<td>4 16</td>
</tr>
<tr>
<td>integer_store</td>
<td>5 18</td>
</tr>
<tr>
<td>iter</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>loop_end</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td></td>
</tr>
<tr>
<td>or_else</td>
<td></td>
</tr>
<tr>
<td>package_elaboration</td>
<td></td>
</tr>
<tr>
<td>procedure_call</td>
<td>9</td>
</tr>
<tr>
<td>procedure_end</td>
<td>28 100</td>
</tr>
<tr>
<td>queue_entry</td>
<td>992 2082</td>
</tr>
<tr>
<td>range_check</td>
<td></td>
</tr>
<tr>
<td>real_literal_access</td>
<td></td>
</tr>
<tr>
<td>record</td>
<td></td>
</tr>
<tr>
<td>record_aggr_comp_access</td>
<td></td>
</tr>
<tr>
<td>record_aggr_setup</td>
<td></td>
</tr>
<tr>
<td>record_comp_access</td>
<td></td>
</tr>
<tr>
<td>record_comp_store</td>
<td></td>
</tr>
<tr>
<td>rendezvous</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td></td>
</tr>
<tr>
<td>slice_access</td>
<td></td>
</tr>
<tr>
<td>slice_store</td>
<td></td>
</tr>
<tr>
<td>string_literal_access</td>
<td></td>
</tr>
<tr>
<td>task_body_elab</td>
<td>92 158</td>
</tr>
<tr>
<td>task_spec_elab</td>
<td>2202 4166</td>
</tr>
<tr>
<td>terminate</td>
<td>24 94</td>
</tr>
<tr>
<td>timed_entry</td>
<td></td>
</tr>
<tr>
<td>variant_tag_check</td>
<td></td>
</tr>
</tbody>
</table>

---

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_sharing 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;
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
TEXTIO.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 interfere 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_DURATE : 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 ;

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.ITERATIONCOUNT := I ;
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.ITERATIONCOUNT := 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 ;
    else
        end if ;
    end if ;
end if;

-- MINIMUM_TIME is now the larger of 1.0 second,
-- 100*SYSTEM.TICK,
-- 100*DURATION\$MIN

CONTROL_DURATION := 0.0 ;
WALL_CONTROL_DURATION := 0.0 ;

end ITERATION ;
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 definitions

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;
  end case;
end record;
when others =>
  char_comp_1, character;
  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_1dim_integer;
                    array_par_in_out_2: in out array_2dim_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
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
procedure proc_1(pointer_par_in: in record_pointer) is -- executed once

begin
next_record := pointer_par_in.pointer_comp.all; -- pointer_glob.next.all
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in.pointer_comp.all;
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in.enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointer_par_in enum_comp.next)
next_record := pointer_par_in(pointe...
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;
        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 + 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: string;
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.

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 regarding 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(DIMENSIONAL_CHECK);
-- pragma SUPPRESS(INDEX_CHECK);
-- pragma SUPPRESS(LENGTH_CHECK);
-- pragma SUPPRESS(RANGE_CHECK);
-- pragma SUPPRESS(DIVISION_CHECK);
-- pragma SUPPRESS(USEDBUG_CHECK);
-- pragma SUPPRESS(STORAGECHECK);
-- pragma SUPPRESS(ELABORATIONCHECK);
package REALIO is new FLOATIO(FLOAT); use REALIO;

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(l):=(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 PO is
-- tests computations with no parameters
-- T,T2 : FLOAT are global
-- El : VECTOR(1..4) is global
-- J,K,L : INTEGER are global
begin
    El(J) := El(K);
    El(K) := El(J);
    El(J) := El(K);
end PO;

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
    El(1) := 1.0;
    El(2) := -1.0;
    El(3) := -1.0;
    El(4) := -1.0;
    for I in 1..N2 loop
        El(1) := (El(1) + El(2) + El(3) - El(4)) * T;
        El(2) := (El(1) + El(2) - El(3) + El(4)) * T;
        El(3) := (El(1) - El(2) + El(3) + El(4)) * T;
        El(4) := (-El(1) + El(2) + El(3) + El(4)) * T;
    end loop;
    -- end Module 2

    START_TIME := FLOAT(CPU_TIME_CLOCK); -- Get Whetstone start time
    CYCLE_LOOP:
end for
end CYCLE_LOOP;
Module 3: passing an array as a parameter

```
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(1+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 parameter

```
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
-- 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;
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 Module 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 procedure
  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 milliseconds 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) & " KIPS");
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 PING_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 environment 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 );
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
--
-- 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_COUNT ; -- 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 environment 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 ) ;
end loop ;

-- 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 PING.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.S friend.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'(0:3161912115 => BIT_VALUE_1,
115/711113 => BIT_VALUE_2,
others => BIT_VALUE_3);

UNPACKED_2 := 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 environment 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;
    REMOTE;
  end loop;
end loop;

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 P00001 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 environment 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;
87

-- 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 PWGIO ; -- output results

procedure P0000010 is -- main procedure to execute
    CPU_TIME : DURATION ; -- CPU time for one feature execution
    WALL_TIME : DURATION ; -- WALL time for one feature execution
    ITERATION_COUNT : ITERATION.ITERATION_COUNTh ; -- 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 environment 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 T0000001 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 0.5 ; -- wait for stable environment 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 ) ;
    end loop ;

    -- 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.PING_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");

abert 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 PING_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 environment 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 ;
    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.

PO000001:

```
1FC910A_10C78: [DN_LOOP
  lx_line_count = 7
] 1FC910A_10E82: [DN_FOR
  [DN_ITERATION_ID
    SM_SEQNUM = 1
    SM_PARENT = [DN_PROC_ID ^]
    lx_symrep = "I"
    sm_obj_type = [DN_RANGE ^]
  ]
  1FC910A_11018: [DN_RANGE
    sm_base_type = [DN_INTEGER ^]
    [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_STMT
  lx_line_count = 5
```
1FC910A_111F3:

[DN_ASSIGN
  lx_used_object_id
  [DN_USED_OBJECT_ID
    lx_symrep = "GLOBAL"
    sm_defn = [DN_VAR_ID ^]
    sm_exp_type = [DN_CONSTRANGED ^]
    sm_value = Uninitialized
  ]

1FC910A_11270:

[DN_NUMERIC_LITERAL
  lx_numrep = "0"
  sm_exp_type = [DN_INTEGER ^]
  sm_value = 0
]

1FC910A_1131F:

] [DN_LOOP
  lx_line_count = 4
  [DN_FOR
    [DN_ITERATION_ID
      SM_SEQNUM = 1
      SM_PARENT = [DN_PROC_ID ^]
      lx_symrep = "INSIDE_LOOP"
      sm_obj_type = [DN_RANGE ^]
    ]
    [DN_RANGE
      sm_base_type = [DN_INTEGER ^]
      [DN_NUMERIC_LITERAL
        lx_numrep = "1"  
        sm_exp_type = [DN_INTEGER ^]
        sm_value = 1
      ]
      [DN_USED_OBJECT_ID
        lx_symrep = "CHECK_TIMES"
        sm_defn = [DN_NUMBER_ID ^]
        sm_exp_type = [DN_INTEGER ^]
        sm_value = 100
      ]
    ]
  ]
  [DN_STMTS
    lx_line_count = 2
    [DN_ASSIGN
      lx_used_object_id
      [DN_USED_OBJECT_ID
        lx_symrep = "GLOBAL"
        sm_defn = [DN_VAR_ID ^]
        sm_exp_type = [DN_CONSTRANGED ^]
        sm_value = Uninitialized
      ]
      [DN_FUNCTION_CALL
        sm_exp_type = [DN_INTEGER ^]
        sm_value = Uninitialized
        sm_normalized_param_s = [DN_EXP_S ^]
        lx_prefix = FALSE
        [DN_USED_BLTN_OP
          SM_ORIGINAL_NODE = [DN_USED_OP ^]
          lx_symrep = "+"
          sm_operator = INTEGER_ADD
        ]
      ]
      [DN_PARAM_ASSOC_S
        [DN_USED_OBJECT_ID
          lx_symrep = "GLOBAL"
          sm_defn = [DN_VAR_ID ^]
          sm_exp_type = [DN_CONSTRANGED ^]
          sm_value = Uninitialized
        ]
        [DN_USED_OBJECT_ID
          lx_symrep = "A_ONE"
          sm_defn = [DN_VAR_ID ^]
          sm_exp_type = [DN_CONSTRANGED ^]
          sm_value = Uninitialized
        ]
      ]
    ]
  ]
]
1FC910A_11DF7:
    [DN_PROCEDURE_CALL
      lx_line_count = 1
      sm_normalized_param_s = [DN_EXP_S ^]
      [DN_USED_NAME_ID
        lx_symrep = "REMOTE"
        sm_defn = [DN_PROC_ID ^]
      ]
    [DN_PARAM_ASSOC_S]

1FC910A_11E98:

1FC910A_11F12:
SimpleTasks:
1F770A_AD12: [ON_COMP_UNIT]
  lx_line_count = 55
  SM_ID_TABLE =
1F770A_CE56: [ON_CONTEXT]
  lx_line_count = 0
}
1F770A_CE6: [ON_SUBPROGRAM_BODY]
  SM_FORWARD = [ON_SUBPROGRAM_DECL ^]
  lx_line_count = 55
1F770A_CF6A: [ON_PROC_ID]
  DO_TRUST_ME = 2
  lx_symrepl = "DOIT"
  sm_spec = [ON_PROCEDURE ^]
  sm_body = [ON_BLOCK ^]
  sm_stub = null
  sm_first = [ON_PROC_ID ^]
1F770A_DB0C: [ON_PROCEDURE]
1F770A_D10B: [ON_PARAMS]
  lx_line_count = 0
  SM_ID_TABLE =
}
1F770A_D19B: [ON_BLOCK]
  POST_COMMENT_HEIGHT = -2
  PRE_COMMENT_HEIGHT = -2
  lx_line_count = 53
1F770A_13074: [ON_ITEM S]
  lx_line_count = 49
1F770A_D25A: [ON_VAR]
  lx_line_count = 1
1F770A_DO54: [ON_VAR_ID]
1F770A_D2FE: [ON_VAR_ID]
  SM_PARENT = [ON_PROC_ID ^]
  lx_symrepl = "COUNT_A"
  sm_obj_type = [ON_CONSTRAINED ^]
  sm_obj_def = [ON_NUMERIC_LITERAL ^]
1F770A_D429: [ON_VAR_ID]
  SM_PARENT = [ON_PROC_ID ^]
  lx_symrepl = "COUNT_B"
  sm_obj_type = [ON_CONSTRAINED ^]
  sm_obj_def = [ON_NUMERIC_LITERAL ^]
}
1F770A_D5B3: [ON_CONSTRAINED]
  sm_type_struct = [ON_INTEGER ^]
  sm_base_type = [ON_INTEGER ^]
  sm_constraint = [ON_RANGE ^]
1F770A_D695: [ON_USED_NAME_ID]
  lx_symrepl = "INTEGER"
  sm_defn = [ON_TYPE_ID ^]
  [ON VOID]
1F770A_D70C: [ON_NUMERIC_LITERAL]
  lx_symrepl = "8"
  sm_exp_type = [ON_INTEGER ^]
  sm_value = 0
}
1F770A_D7A6: [ON_VAR]
  POST_COMMENT_HEIGHT = -2
  lx_line_count = 2
1F770A_D975: [ON_VAR_ID]
1F770A_D84A: [ON_VAR_ID]
  SM_PARENT = [ON_PROC_ID ^]
  lx_symrepl = "RESULT_C"
  sm_obj_type = [ON_CONSTRAINED ^]
  sm_obj_def = [ON_NUMERIC_LITERAL ^]
}
1F770A_D904: [ON_CONSTRAINED]
  sm_type_struct = [ON_INTEGER ^]
  sm_base_type = [ON_INTEGER ^]
  sm_constraint = [ON_RANGE ^]
1F770A_DAB6: [ON_USED_NAME_ID]
  lx_symrepl = "INTEGER"
  sm_defn = [ON_TYPE_ID ^]
  [ON VOID]
1F770A_DB2D: [ON_NUMERIC_LITERAL]
  lx_symrepl = "6"
  sm_exp_type = [ON_INTEGER ^]
  sm_value = 6
}
1F770A_DC0A: [ON_TASKDECL]
  POST_COMMENT_HEIGHT = -2
97

1F770A Dü94:

[ON_VAR_ID
SM_PARENT = [ON_PROC_ID ^]
lx_symrep = "TASK_A"
sm_obj_type = [ON_TASK_SPEC ^]
sm_obj_def = [ON_TASK_SPEC ^]
]

1F770A dü08A:

[ON_TASK_SPEC
lx_line_count = 0
sm_body = [ON_BLOCK ^]
]

1F770A dü10D:

[ON_DECL_S
lxVERBOSE = TRUE
lx_line_count = 0
SM_ID_TABLE =
]

1F770A dü3FF:

[ON_TASK_DECL
POST_COMMENT_HEIGHT = -2
lx_line_count = 3
[ON_VAR_ID
SM_PARENT = [ON_PROC_ID ^]
lx_symrep = "TASK_B"
sm_obj_type = [ON_TASK_SPEC ^]
sm_obj_def = [ON_TASK_SPEC ^]
]

1F770A dü50F:

[ON_TASK_SPEC
lx_line_count = 0
sm_body = [ON_BLOCK ^]
]

1F770A dü5E6:

[ON_TASK_DECL
POST_COMMENT_HEIGHT = -2
lx_line_count = 6
[ON_VAR_ID
SM_PARENT = [ON_PROC_ID ^]
lx_symrep = "TASK_C"
sm_obj_type = [ON_TASK_SPEC ^]
sm_obj_def = [ON_TASK_SPEC ^]
]

1F770A dü5F4:

[ON_TASK_SPEC
lx_line_count = 0
sm_body = [ON_BLOCK ^]
]

1F770A dü6C0:

[ON_DECL_S
lx_line_count = 3
[ON_SUBPROGRAM_DECL
lx_line_count = 1
[ON_ENTRY_ID
SM_SEQNUM = 1
SM_PARENT = [ON_VAR_ID ^]
lx_symrep = "ENTRY_A"
sm_spec = [ON_ENTRY ^]
]

1F770A dü64B:

[ON_ENTRY
lx_line_count = 0
[ON_VOID]
]

1F770A dü605:

[ON_PARAM_S
lx_line_count = 0
SM_ID_TABLE =
]

1F770A dü79A:

[ON_SUBPROGRAM_DECL
lx_line_count = 1
[ON_ENTRY_ID
SM_SEQNUM = 2
SM_PARENT = [ON_VAR_ID ^]
lx_symrep = "ENTRY_B"
sm_spec = [ON_ENTRY ^]
]

1F770A dü95E:

[ON_ENTRY
lx_line_count = 0
[ON_VOID]
]

1F770A dü9E8:

[ON_PARAM_S
lx_line_count = 0
SM_ID_TABLE =
]

1F770A düaAD:

[ON_SUBPROGRAMDECL
lx_line_count = 1
[ON_ENTRY_ID
SM_SEQNUM = 3
SM_PARENT = [ON_VAR_ID ^]
lx_symrep = "DONE"
sm_spec = [ON_ENTRY ^]
]


```
1F770DA.F471:
  [ON_ENTRY
   lx_line_count = 0
  [ON_VOID]

1F770DA.F47B:
  [ON_PARAM
   lx_line_count = 0
   SM_ID_TABLE =
   ]

1F770DA.F481:
  [ON_TASK
   SM_FORWARD = [ON_TASK_DECL ^]
   lx_line_count = 9
   [ON_TASK_BODY
    lx_symrep = "TASK_A"
    sm_spec = [ON_TASK_SPEC ^]
    sm_body = [ON_BLOCK ^]
    sm_stub = null
    sm_first = [ON_VAR_ID ^]
   ]

1F770DA.F48C:
  [ON_BLOCK
   POST_COMMENT_HEIGHT = -2
   lx_line_count = 8
   [ON_ITEM
    lx_line_count = 0
    SM_ID_TABLE =
   ]

1F770DA.F490:
  [ON_STMT
   lx_line_count = 6
   [ON_LOOP
    lx_line_count = 3
   ]

1F770DA.F494:
  [ON_ITERATION
   SM_SEQNUM = 1
   SM_PARENT = [ON_VAR_ID ^]
   lx_symrep = "P"
   sm_obj_type = [ON_RANGE ^]

1F770DA.F498:
  [ON_RANGE
   sm_base_type = [ON.INTEGER ^]
   [ON_NUMERIC_LITERAL
    lx_numrep = "1"
    sm_exp_type = [ON.INTEGER ^]
    sm_value = 1
   ]

1F770DA.F49D:
  [ON_NUMERIC_LITERAL
   SM_INT_VALUE = 1600
   lx_numrep = "1000"
   sm_exp_type = [ON.INTEGER ^]
   sm_value = 1600
  ]

1F770DA.F4A2:
  [ON_STMT
   lx_line_count = 1

1F770DA.F4A4:
  [ON_ASSIGN
   lx_line_count = 1
   [ON_USED_OBJECT_ID
    lx_symrep = "COUNT_A"
    sm_defn = [ON_VAR_ID ^]
    sm_exp_type = [ON_CONSTRAINED ^]
    sm_value = Uninitialized
   ]

1F770DA.F4B0:
  [ON_FUNCTION
    sm_exp_type = [ON.INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [ON_EXP_S ^]
    lx_prefix = FALSE
    [ON_USED_BUILTIN
     SM_ORIGINAL_NODE = [ON_USED.OP ^]
     lx_symrep = "+
     sm_operator = INTEGER_ADD
    ]

1F770DA.F4B7:
  [ON_PARAM
   [ON_USED_OBJECT_ID
    lx_symrep = "COUNT_A"
    sm_defn = [ON_VAR_ID ^]
    sm_exp_type = [ON_CONSTRAINED ^]
    sm_value = Uninitialized
   ]

1F770DA.F4C4:
  [ON_NUMERIC_LITERAL
    lx_numrep = "1"
    sm_exp_type = [ON.INTEGER ^]
    sm_value = 1
  ]

1F770DA.F4D2:
  [ON_ENTRY_CALL
```
SM_ORIGINATED_NODE = [DN_PROCEDURE_CALL]
lx_line_count = 1
sm_normalized_params = [DN_EXP_S]
[DN_SELECTED]
sm_exp_type = null
sm_value = uninitialized
[DN_USED_OBJECT_ID]
lx_symrep = "TASK_A"
sm_defn = [DN_VAR_ID]
sm_exp_type = [DN_TASK_SPEC]
sm_value = uninitialized
[DN_USED_NAME_ID]
lx_symrep = "ENTRY_A"
sm_defn = [DN_ENTRY_ID]
[DN_PARAM_ASSOC_S]

[DN_ASSIGN]
lx_line_count = 1
[DN_USED_OBJECT_ID]
lx_symrep = "COUNT_A"
sm_defn = [DN_VAR_ID]
sm_exp_type = [DN_CONSTRAINED]
sm_value = uninitialized
[DN_FUNCTION_CALL]
sm_exp_type = [DN_INTEGER]
sm_value = uninitialized
sm_normalized_params = [DN_EXP_S]
lx_prefix = FALSE
[DN_USED_BLTN_OP]
sm_original_node = [DN_USED_OP]

[DN_PARAM_ASSOC_S]

[DN_PARAM_ASSOC_S]

[DN_ALTERNATIVE_S]
lx_line_count = 0

[DN_TKAN_BODY]
sm_forward = [DN_TASK_DECL]
lx_line_count = 9
[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]

[DN_BLOCK]
POST_COMMENT_HEIGHT = -2
lx_line_count = 8
[DN_ITEM_S]
lx_line_count = 8
SM_ID_TABLE =

[DN_STMT_S]
lx_line_count = 6

[DN_LOOP]
lx_line_count = 3

[DN_FOR]
[DN_ITERATION_ID]
sm_seqnum = 1
sm_parent = [DN_VAR_ID]
lx_symrep = "I"
sm_obj_type = [DN_RANGE]

[DN_RANGE]
sm_base_type = [DN_INTEGER]
sm_numrep = "1"
sm_exp_type = [DN_INTEGER]
sm_value = 1

100

1F77DA_11014:
  [DN_NUMERIC_LITERAL
    lx_numrep = "100"
    sm_exp_type = [DN_INTEGER ^]
    sm_value = 100
  ]

1F77DA_11559:
  [DN_STMT]
  lx_line_count = 1
  [DN_ASSIGN]
    lx_line_count = 1

1F77DA_11138:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_111DA:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_11423:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN.INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_114FA:
  [DN_PARAM_ASSOC_S
    [DN_USED_OBJECT_ID
      lx_symrep = "COUNT_B"
      sm_defn = [DN_VAR_ID ^]
      sm_exp_type = [DN_CONSTRAINED ^]
      sm_value = Uninitialized
    ]
  ]

1F77DA_1134C:
  [DN_PARAM_ASSOC_S
    [DN_USED_OBJECT_ID
      lx_symrep = "COUNT_B"
      sm_defn = [DN_VAR_ID ^]
      sm_exp_type = [DN_CONSTRAINED ^]
      sm_value = Uninitialized
    ]
  ]

1F77DA_115C9:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_PROCEDURE_CALL ^]
    lx_line_count = 1
    sm_normalized_param_s = [DN_EXP_S ^]
  ]

1F77DA_1156D:
  [DN_SELECTED
    sm_exp_type = null
    sm_value = Uninitialized
  ]

1F77DA_11711:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_11783:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_1182A:
  [DN_PARAM_ASSOC_S
    [DN_USED_OBJECT_ID
      lx_symrep = "ENTRY_B"
      sm_defn = [DN_ENTRY_ID ^]
    ]
  ]

1F77DA_1192F:
  [DN_ASSIGN
    lx_line_count = 1
    [DN_USED_OBJECT_ID
      lx_symrep = "COUNT_B"
      sm_defn = [DN_VAR_ID ^]
      sm_exp_type = [DN_CONSTRAINED ^]
      sm_value = Uninitialized
    ]
  ]

1F77DA_11A58:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_11B1A:
  [DN_FUNCTION_CALL
    sm_exp_type = [DN_INTEGER ^]
    sm_value = Uninitialized
    sm_normalized_param_s = [DN_EXP_S ^]
    lx_prefix = FALSE
  ]

1F77DA_11D78:
  [DN_PARAM_ASSOC_S
    [DN_USED_OBJECT_ID
      lx_symrep = "COUNT_B"
      sm_defn = [DN_VAR_ID ^]
      sm_exp_type = [DN_CONSTRAINED ^]
      sm_value = Uninitialized
    ]
  ]

1F77DA_11C4A:
  [DN_USED_OBJECT_ID
    lx_symrep = "COUNT_B"
    sm_defn = [DN_VAR_ID ^]
    sm_exp_type = [DN_CONSTRAINED ^]
  ]
sm_value = Uninitialized

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP

1F770A_11E4A:
[DN_ALTERNATIVE_S

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP

1F770A_11F88:
[DN_TASK_BODY

1F770A_12012:
[DN_TASK_BODY_ID

1F770A_12101:
[DN_BLOCK

1F770A_121A5:
[DN_ITEM_S

1F770A_12804:
[DN_ACCEPT

1F770A_12824:
[DN_PARAM_S

1F770A_12978:
[DN_USED_NAME_ID

1F770A_1299F:
[DN_PARAM_S

1F770A_12FA9:
[DN_ASSIGN

1F770A_12FC6:
[DN_FUNCTION_CALL

1F770A_12C05:
[DN_USED_BLK_OP
1F7700A_13C36:
  [DN_ALTERNATIVE_S
   lx_line_count = 0
  ]

1F7700A_14630:
  [DN_STMT_S
   PRE_COMMENT_HEIGHT = 1
   lx_line_count = 2
  ]

1F7700A_142F2:
  [DN_ENTRY_CALL
   SM_ORIGINAL_NODE = [DN_PROCEDURE_CALL ^]
   lx_line_count = 2
   sm_normalized_param_s = [DN_EXP_S ^]
  ]

1F7700A_14506:
  [DN_SELECTED
   sm_exp_type = null
   sm_value = Uninitialized
  ]

1F7700A_1443A:
  [DN_USED_OBJECT_ID
   lx_symrep = "TASK_C"
   sm_defn = [DN_VAR_ID ^]
   sm_exp_type = [DN_TASK_SPEC ^]
   sm_value = Uninitialized
  ]

1F7700A_1440C:
  [DN_USED_NAME_ID
   lx_symrep = "DONE"
   sm_defn = [DN_ENTRY_ID ^]
  ]

1F7700A_14553:
  [DN_PARAM_ASSOC_S]

1F7700A_146F0:
  [DN_ALTERNATIVE_S
   lx_line_count = 0
  ]

1F7700A_1483C:
  [DNPragma_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 14.3833 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.
Test Name: CPU00090

Test Description: Determine clock resolution using second differences of values returned by the function CPU_Time_Clock.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Test Iteration 1</th>
<th>Test Iteration 2</th>
<th>Test Iteration 3</th>
<th>Test Iteration 4</th>
<th>Test Iteration 5</th>
<th>Test Iteration 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>73770.000</td>
<td>73771.000</td>
<td>73772.000</td>
<td>73773.000</td>
<td>73774.000</td>
<td>73775.000</td>
</tr>
<tr>
<td>2</td>
<td>73771.000</td>
<td>73772.000</td>
<td>73773.000</td>
<td>73774.000</td>
<td>73775.000</td>
<td>73776.000</td>
</tr>
<tr>
<td>3</td>
<td>73772.000</td>
<td>73773.000</td>
<td>73774.000</td>
<td>73775.000</td>
<td>73776.000</td>
<td>73777.000</td>
</tr>
<tr>
<td>4</td>
<td>73773.000</td>
<td>73774.000</td>
<td>73775.000</td>
<td>73776.000</td>
<td>73777.000</td>
<td>73778.000</td>
</tr>
<tr>
<td>5</td>
<td>73774.000</td>
<td>73775.000</td>
<td>73776.000</td>
<td>73777.000</td>
<td>73778.000</td>
<td>73779.000</td>
</tr>
</tbody>
</table>

Average time per cycle: 2784.3 milliseconds

Average Whetstone rating: 359 MHz/GHz

Test Name: CPU00091

Class Name: Composite

Test Description: Reidok F. Meister's GHUSTONE composite benchmark

Test Name: CPU00092

Class Name: Composite

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00093

Class Name: Chapter 13

Test Description: Time to perform standard boolean operations on arrays of booleans.

Test Name: CPU00094

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00095

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00096

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00097

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00098

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00099

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00100

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00101

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00102

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00103

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating

Test Name: CPU00104

Class Name: Chapter 13

Test Description: CPU cycles per Whetstone rating
null
<table>
<thead>
<tr>
<th>Iteration</th>
<th>78703.0000 78703.0000</th>
<th>0.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 1</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 2</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 3</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 4</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 5</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 6</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 7</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 8</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 9</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 10</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 11</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 12</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 13</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 14</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 15</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 16</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 17</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 18</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 19</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 20</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 21</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 22</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 23</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 24</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 25</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 26</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 27</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 28</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 29</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 30</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 31</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 32</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 33</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 34</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 35</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 36</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 37</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 38</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 39</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 40</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 41</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 42</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 43</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 44</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 45</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 46</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 47</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 48</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 49</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 50</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 51</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 52</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 53</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 54</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 55</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 56</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 57</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 58</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 59</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 60</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 61</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 62</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 63</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Test Iteration 64</td>
<td>78703.0000 78703.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

For this test, some tests were not performed on components in a loop.
Iteration 8
Test Name: C000002
CPU Time: 99.1 microseconds
Wall Time: 99.1 microseconds
Iteration Count: 1024
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
Test Name: C00002
CPU Time: 1094.7 microseconds
Wall Time: 1094.7 microseconds
Iteration Count: 1024
Description: Task create and terminate 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
Test Name: P000010
Class Name: Procedure
CPU Time: 15.9 microseconds
Wall Time: 15.9 microseconds
Iteration Count: 16384
Test Description: Compare to P000005
10 parameters, in INTEGER

Iteration 1
Test Name: C000003
Class Name: Tasking
CPU Time: 254.1 microseconds
Wall Time: 254.1 microseconds
Iteration Count: 2048
Test Description: Minimum measurement, entry call and return time
1 task entry, task inside procedure

Iteration 1
Test Name: P000008
Class Name: Tasking
CPU Time: 371.3 microseconds
Wall Time: 371.3 microseconds
Iteration Count: 1024
Test Description: Task entry call and return time measured
One task active, two entries, two tasks in a package
using select statement.

Test Name: A000091
Class Name: Composite
1.2000 in time in milliseconds for one iteration

Test Name: A000092
Class Name: Composite
Average time per cycle: 2864.00 milliseconds
Average Whetstone rating : 357 MFLOPS

Iteration 1
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
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td>4</td>
<td>0.000</td>
</tr>
<tr>
<td>8</td>
<td>0.000</td>
</tr>
<tr>
<td>16</td>
<td>0.000</td>
</tr>
<tr>
<td>32</td>
<td>0.000</td>
</tr>
<tr>
<td>64</td>
<td>0.000</td>
</tr>
<tr>
<td>128</td>
<td>0.000</td>
</tr>
<tr>
<td>256</td>
<td>0.000</td>
</tr>
<tr>
<td>512</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**Test Name:** | **Class Name:** |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PH000080</td>
<td>Procedure</td>
</tr>
<tr>
<td>PH000040</td>
<td>Tasking</td>
</tr>
<tr>
<td>PH000009</td>
<td>Tasking</td>
</tr>
</tbody>
</table>

**CPU Time:** | 17.7 microseconds
**Wall Time:** | 17.7 microseconds
**Iteration Count:** | 16384
**Test Description:**
Procedure call and return time measurement

**Tests:**
- Test 1: 0.000
- Test 2: 0.000
- Test 4: 0.000
- Test 8: 0.000
- Test 16: 0.000
- Test 32: 0.000
- Test 64: 0.000
- Test 128: 0.000
- Test 256: 0.000
- Test 512: 0.000
- Test 1024: 0.000
- Test 2048: 0.000
- Test 4096: 0.000
- Test 8192: 0.000
- Test 16384: 0.000

**CPU Time:** | 17.7 microseconds
**Wall Time:** | 17.7 microseconds
**Iteration Count:** | 1024
**Test Description:**
Task creates and terminates measurement
- with one task, no entries when task is in a procedure, using a task type in a package, no select statement, no loop

**Tests:**
- Test 1: 0.000
- Test 2: 0.000
- Test 4: 0.000
- Test 8: 0.000
- Test 16: 0.000
- Test 32: 0.000
- Test 64: 0.000
- Test 128: 0.000
- Test 256: 0.000
- Test 512: 0.000
- Test 1024: 0.000
- Test 2048: 0.000
- Test 4096: 0.000
- Test 8192: 0.000
- Test 16384: 0.000

**CPU Time:** | 106.5 microseconds
**Wall Time:** | 106.5 microseconds
**Iteration Count:** | 1024
**Test Description:**
Task creates and terminates measurement
- with one task, no entries when task is in a procedure, task defined and used in procedure, no select statement, no loop

**Tests:**
- Test 1: 0.000
- Test 2: 0.000
- Test 4: 0.000
- Test 8: 0.000
- Test 16: 0.000
- Test 32: 0.000
- Test 64: 0.000
- Test 128: 0.000
- Test 256: 0.000
- Test 512: 0.000
- Test 1024: 0.000
- Test 2048: 0.000
- Test 4096: 0.000
- Test 8192: 0.000
- Test 16384: 0.000

**CPU Time:** | 465.2 microseconds
**Wall Time:** | 465.2 microseconds
**Iteration Count:** | 2048
**Test Description:**
Time to perform standard boolean operations on arrays of Booleans.

**Tests:**
- Test 1: 0.000
- Test 2: 0.000
- Test 4: 0.000
- Test 8: 0.000
- Test 16: 0.000
- Test 32: 0.000
- Test 64: 0.000
- Test 128: 0.000
- Test 256: 0.000
- Test 512: 0.000
- Test 1024: 0.000
- Test 2048: 0.000
- Test 4096: 0.000
- Test 8192: 0.000
- Test 16384: 0.000

**CPU Time:** | 7.9 microseconds
**Wall Time:** | 7.9 microseconds
**Iteration Count:** | 110
**Test Description:**
Rejects P. Mahony's DRYSTONE compiler benchmark

**Tests:**
- Test 1: 0.000
- Test 2: 0.000
- Test 4: 0.000
- Test 8: 0.000
- Test 16: 0.000
- Test 32: 0.000
- Test 64: 0.000
- Test 128: 0.000
- Test 256: 0.000
- Test 512: 0.000
- Test 1024: 0.000
- Test 2048: 0.000
- Test 4096: 0.000
- Test 8192: 0.000
- Test 16384: 0.000

**Average time per cycle:** 2.474.47 milliseconds
**Average Whetstone rating:** 235 MIPS
112

Iteration 16384
1998.0000 2093.0000 95.0000
Test Name: P000001
Class Name: Procedure
CPU Time: 4.3 microseconds
Hall Time: 4.3 microseconds
Test Description:
Procedure call and return time (may be zero if automatic inlining).
Procedure has local no parameters.

Iteration 1
1997.0000 2197.0000 0.0000
Iteration 2
1996.0000 2196.0000 0.0000
Iteration 4
1996.0000 2196.0000 0.0000
Iteration 8
1996.0000 2196.0000 0.0000
Iteration 16
1996.0000 2199.0000 1.0000
Iteration 32
1999.0000 2199.0000 0.0000
Iteration 64
2000.0000 2200.0000 0.0000
Iteration 128
2000.0000 2202.0000 1.0000
Iteration 256
2200.0000 2204.0000 1.0000
Iteration 512
2204.0000 2210.0000 2.0000
Iteration 1024
2210.0000 2211.0000 3.0000
Iteration 2048
2211.0000 2212.0000 12.0000
Iteration 4096
2212.0000 2214.0000 26.0000
Iteration 8192
2214.0000 2219.0000 52.0000
Iteration 16384
2219.0000 2234.0000 103.0000

Test Name: P000010
Class Name: Procedure
CPU Time: 18.1 microseconds
Hall Time: 18.1 microseconds
Iteration Count: 16384

Test Description:
Procedure call and return time measurement.
Compare to P000001
10 parameters, in INT32.

Iteration 1
2640.0000 2840.0000 0.0000
Iteration 2
2641.0000 2841.0000 0.0000
Iteration 4
2641.0000 2841.0000 0.0000
Iteration 8
2641.0000 2841.0000 0.0000
Iteration 16
2641.0000 2841.0000 0.0000
Iteration 32
2641.0000 2841.0000 0.0000
Iteration 64
2641.0000 2841.0000 0.0000
Iteration 128
2641.0000 2841.0000 0.0000
Iteration 256
2641.0000 2841.0000 0.0000
Iteration 512
2641.0000 2841.0000 0.0000
Iteration 1024
2641.0000 2841.0000 0.0000
Iteration 2048
2641.0000 2841.0000 0.0000
Iteration 4096
2641.0000 2841.0000 0.0000
Iteration 8192
2641.0000 2841.0000 0.0000
Iteration 16384
2641.0000 2841.0000 0.0000

Test Name: T000001
Class Name: Tasking
CPU Time: 474.4 microseconds
Hall Time: 474.4 microseconds
Iteration Count: 2048

Test Description:
Miscellaneous function, entry call and return time.
1 task 1 entry, task inside procedure.
no select

Iteration 1
2008.0000 2008.0000 0.0000
Iteration 2
2008.0000 2008.0000 0.0000
Iteration 4
2008.0000 2008.0000 0.0000
Iteration 8
2008.0000 2008.0000 0.0000
Iteration 16
2008.0000 2008.0000 0.0000
Iteration 32
2008.0000 2008.0000 0.0000
Iteration 64
2008.0000 2008.0000 0.0000
Iteration 128
2008.0000 2008.0000 0.0000
Iteration 256
2008.0000 2008.0000 0.0000
Iteration 512
2008.0000 2008.0000 0.0000
Iteration 1024
2008.0000 2008.0000 0.0000
Iteration 2048
2008.0000 2008.0000 0.0000
Iteration 4096
2008.0000 2008.0000 0.0000
Iteration 8192
2008.0000 2008.0000 0.0000
Iteration 16384
2008.0000 2008.0000 0.0000

Test Name: T000002
Class Name: Tasking
CPU Time: 144.4 microseconds
Hall Time: 144.4 microseconds
Iteration Count: 1024

Test Description:
Task entry call and return time measured.
1 task 2 entries, task in a package.
using select statement.


