Chapter 1

Introduction

Russell M. Clapp Trevor Mudge

The University of Michigan

The ACM Special Interest Group on Ada (SIGAda) Performance Issues Working Group (PIWG) was formed in November 1984. The charter of PIWG is to provide the Ada community with performance related information. In 198 benchmarks to measure the speed and capacity of Ada compiler systems and identify associated performance issues. These issues vary in nature, but all are directly associated with Ada compilers and their run-time systems. In order to identify and examine these issues, an approach is needed that can be applied to all Ada compiler systems.

The approach taken by PIWG to identify and evaluate Ada performance issues is made up of two major facets. One is an analysis of the features and functionality that the language standard requires of an implementation. The second is the development of tools designed to evaluate the performance of these features in a manner that is as implementation independent as possible.

An analysis of compiler system requirements to support Ada uncovers a large number of issues regarding performance. Since we are initially only concerned with the performance evaluation of compiler implementations themselves (and not, for example, with evaluation of the ease of use of a development environment), the issues considered are focused on the time and space requirements of an Ada compiler at both compile time and run time. Further discussion of these issues including a taxonomy of the compiler characteristics to be addressed is given in Chap. 2 below. Details regarding the time and space issues are discussed in Chaps. 3 and 4 respectively.

As for the evaluation tools themselves, the initial method employed by PIWG is the use of benchmark programs. These programs are written in Ada to provide an implementation independent and portable way of measuring performance. The design and implementation of such programs is not a straightforward task, however. There are many factors that may influence the results obtained from benchmark programs, because computer systems are in general a complex configuration of various hardware and software elements. These factors must be carefully considered in the design phase as well as accounted for in the evaluation phase. Since these benchmarking tools are to be applied to many different systems, implementation dependencies will certainly come into play. This necessitates the use of prototyping and an iterative design approach. The goal then, is to develop benchmarking

tools that are largely independent of specific systems while accounting for as many of the general system factors as possible. The bulk of this document discusses our approach toward this goal.

1.1 Philosophy on Benchmarks

Benchmark programs are used by PIWG as performance measuring tools for several reasons. For one, these programs are quite portable, and this allows them to be run by a large group of Ada compiler users. Also, the programs have been designed with standardized output formats and have scripts indicating compilation and execution order. These features make the tests easy to use and facilitates their widespread use. The benchmarks have also been carefully designed so that, by and large, the results they provide are reliable. This allows performance evaluation to begin without relying on vendors' claims. However, as we discuss below, variations can occur in the benchmark results, and consultation with vendors may be necessary too pinpoint the source of any inconsistent behavior. In addition to this type of interaction, the vendors' own experience with running the benchmark programs will help them identify points of inefficiency with their compilers that should be addressed.

Several approaches are possible in the development of benchmark programs. These include benchmarks to measure the overhead associated with a particular language feature, synthetic benchmarks designed to measure the time needed to execute a representative mix of instructions, and sample applications such as sorting programs, system utilities, etc. Currently, the PIWG benchmarking suite consists largely of programs designed to measure the overhead of individual language features. This approach is the most useful for pinpointing particular aspects of compiler inefficiency. Data provided by these benchmarks are useful for programmers of real-time systems where hard deadlines must be met. In addition to feature benchmarks, the PIWG suite also contains some small applications as well as the widely used Whetstone [16] and Dhrystone [44] synthetic benchmarks. These benchmarks were designed to measure run-time performance of a typical scientific computation mix (Whetstone) and a typical systems programming instruction mix (Dhrystone) for sequential language compilers. They provide a yardstick for comparison against compilers of sequential languages (e.g. C and Pascal), but do not address issues concerning real-time applications and programming with concurrency in general. As the PIWG suite is enhanced in the future, it will contain additional synthetic benchmarks designed to evaluate concurrency as well as certain application programs. More details on the contents of the PIWG suite are given in Chap. 7.

Though the benchmark programs are intended to measure performance and identify inefficiencies of Ada compilers, it is important to note that the timing results obtained from these programs actually measure the performance of compiler/operating system/hardware configurations. While an effort is made to eliminate undesired interference of the benchmarks from an operating system, no attempt is made to attribute portions of the overhead to the computer hardware, operating system (if present), or compiler. The effects of the various components of the entire system must be examined on an individual basis. No predetermined formula can accurately attribute overhead costs for various system configurations.

In summary, the PIWG benchmarking suite is intended to be a reliable, portable, easy to use set of programs that measure performance of a wide range of language features and characteristics. These tools are also able to measure performance largely independent of particular compiler implementations, but the results they provide must be interpreted as a measure of the aggregate performance of a particular compiler/operating system/computer implementation. Additional details of the specific

structure and use of the programs is given in later chapters.

1.2 History of Ada Benchmarking

In order to motivate some of the design decisions discussed in this and later chapters, we will give a brief review of other Ada benchmarking efforts. An overview of these efforts with a more detailed description of the tests used can be found in [21]. The major benchmarking efforts listed there are The University of Michigan suite [10], the PIWG suite [40], and the Prototype Ada Compiler Evaluation Capability (ACEC) [27]. In addition to these suites, other efforts are currently underway including development of the ACEC suite [41] and the Software Engineering Institute's (SEI) Ada Embedded Systems Testbed Project [45] (see Chap. 8). Other work in the area has been done in examining issues and approaches to benchmarking [6, 18, 39], developing benchmark programs [33, 46, 28, 8], and using benchmark programs [24, 10, 2, 3, 20, 19].

The University of Michigan test suite is one that concentrates on language and run-time system feature benchmarks. The features measured include subprogram overhead, storage allocation and deallocation, exception propagation and handling, task elaboration, activation and termination, task rendezvous, mathematical operation overhead on objects of types TIME and DURATION, and characteristics of the underlying scheduler with regard to delays. There are several variations of each test, to account for different coding styles, approaches, and variations in capacity requirements.

As part of the work to develop the Michigan suite, much emphasis was placed on the requirements for measurement accuracy and precision, as well as machine independence. As a result, a method for obtaining reliable results using the Ada CLOCK function was developed. This approach requires testing of the CLOCK function overhead and resolution, and benchmarks to provide this data are included in the suite. The information obtained from these tests is examined using a *second differencing* technique described in [10] and Chap. 4 below. This calibration phase provides the iteration parameters required for the rest of the test suite. Further discussion are given in [10] regarding machine dependencies, operating system interference, and the effect of compiler optimizations.

The PIWG suite is similar to Michigan's in that it includes many feature tests. However, the PIWG suite also contains the Whetstone and Dhrystone synthetic benchmarks, some small applications programs, and compilation time tests. Most all of the features measured by the Michigan suite are addressed by the PIWG suite, but with different variations. The approaches used for timing and optimization blocking are somewhat similar to Michigan's, but additional timing is done with machine dependent CPU timers for comparison. The PIWG suite also determines clock resolution as part of each benchmark, which in turn dynamically determines the program's execution time.

The Prototype ACEC tests were mainly a collection of existing programs used for benchmarking. They included tests to measure the speed and capacity of several language features as well as compilation time tests and small application benchmarks.

The development of the ACEC suite was completed in 1988 by Boeing Military Airplane under contract to the U. S. Air Force and is a project independent of the original Prototype ACEC. The current suite is a comprehensive one that includes a large number of tests to measure compilation speed, code size, and execution speed of language features. There are many permutations of the language speed tests that are designed to measure the effects of optimization, degradation, design trade-offs, and kernel efficiency. The suite also includes an analysis tool to aid in the interpretation of

the large amount of data produced by the tests. Chapter 8 contains more details on the ACEC suite.

The work done by the SEI as part of the AEST project has included the evaluation of existing benchmark suites in embedded systems. Several reports have outlined the specific interactions of an embedded system with various benchmark programs [2, 3, 20, 19]. The investigation has uncovered some problems with the benchmarks, particularly with their use on embedded systems. These issues are discussed further in Chaps. 3 and 4 below. Chapter 8 also includes a description of the SEI's Hartstone benchmark which was developed for hard real-time systems.

1.3 Summary and Outline

At this point, PIWG is poised to develop its second generation test suite. The basic design has been used in the first generation suite and the Michigan suite. Evaluation of these suites as well as concepts from the ACEC tests will provide input for the next PIWG suite. A description of the philosophy, motivation, and approach to this task with input from other efforts is described in the following chapters. A taxonomy of language characteristics for benchmarking is presented in Chap. 2. Chapters 3 and 4 address the aspects of time and space that relate to the performance of Ada compilers. Consideration is given to issues arising in parallel and distributed system Ada implementations in Chap. 5. The impact of optimization on performance measurement is given in Chap. 6. Chapter 7 discusses the specifics of the PIWG suite of benchmarks. Chapter 8 summarizes the current state of Ada benchmarking and outlines future directions. Complete papers by contributing authors also appear in the chapters of this document where their topic is most closely related.

Chapter 2

Taxonomy of Benchmarks

Russell M. Clapp Trevor Mudge

The University of Michigan

The purpose of a benchmark taxonomy is to provide a high-level classification of various aspects of compiler performance. This classification is intended to identify the various dimensions of a compiler that need to be benchmarked. The taxonomy we use characterizes compiler performance along three dimensions: time, space, and control [1]. These three areas are used, since implementations of particular compiler and run-time system functions routinely involve tradeoffs along these dimensions. An operation may require less time to execute if more space is available, for example, or the predictability of a scheduling interval may be diminished if less space and execution overhead is dedicated to that task. These dimensions for a benchmarking taxonomy are derived from the suggested taxonomy given in [4]. Although the issues described both here and in the reference are overlapping, the taxonomy given in this document is presented along three uniform dimensions. More detailed discussion and examples demonstrating tradeoffs between time, space, and control are given in sections describing compile-time and run-time aspects below.

In order to discuss these dimensions of performance in an organized manner, further distinctions are made regarding compilers. The first and most important of these classifications is that between compile and run time. This distinction separates measurements of compiler front end execution, optimizers, linkers, and loaders from measurements of generated code, run-time system, and possibly operating system execution. Although the implementation needs and goals of these two major areas are quite different, their performance can still be characterized along the dimensions of time, space, and control.

Another distinction that highlights the tradeoffs made in compiler implementation is that between hosted and embedded systems. Typically, the different approaches are used to implement language features for development environments and real-time systems. For example, compile time for an embedded system compiler may be greater than that for a development environment, but the execution time for the embedded system may be much less. It may also be the case that embedded systems emphasize efficiency in time and control at the cost of space. By considering the tradeoffs involved

[1] Control is used to refer to run-time support mechanisms, e.g., task scheduling algorithm.

in different types of systems, a wide range of language features suitable for benchmarking can be identified. By examining the full set of feature measurements, overall efficiency of a compiler can be determined with considerations made for design tradeoffs.

2.1 Run-Time Benchmarks

The goal of run-time performance benchmarks is to measure the execution efficiency of compiler generated code and the compiler's run-time system. These benchmarks will also measure operating system overhead in cases where it is called directly by the generated code or run-time system. Although the benchmark programs are designed to avoid unnecessary and unwanted operating system interference, there are certain basic run-time functions of the operating system that cannot be avoided in some cases (e.g. virtual memory support). It may also be the case that no operating system is present, and the compiler's run-time system is responsible for supporting all run-time functions. No matter how the support of the run-time environment is implemented, the benchmarks are intended to measure the performance of what is actually available to the user.

Because many different implementation strategies are possible for a compiler's run-time operations, the specific areas to be measured are classified using the abstract categories of time, space, and control. Each of these areas are discussed in the following subsections as they pertain to run-time performance benchmarks. Following these subsections, a discussion of the measurement areas is presented that examines various tradeoffs in design for both hosted and embedded systems.

2.1.1 Time

Two main areas fall under the heading of time related benchmarks for compiler run-time performance. One is the accuracy and precision of timing operations in the language, and the other is the execution speed of object code and run-time support. In addition to evaluating the quality and suitability of timing related operations in the language, the benchmarking of time accuracy and precision also provides vital information regarding the overall benchmarking environment provided at the language level. This information yields values for the parameters of all benchmark tests that involve the measurement of execution time.

2.1.1.1 Accuracy and Precision

The benchmark tests that fall under this heading are those that evaluate the timing function. This is basically an evaluation of the CLOCK function of the CALENDAR package. Of course, this function, to a large extent, reflects the timing environment available in the hardware. However, the evaluation of this function also measures the efficiency with which the hardware is used. This efficiency may be diminished by the use of several layers of software (e.g. the run-time and operating systems) and their overhead costs that limit the amount of practical timing precision available. The accuracy of timing is also affected by this overhead in the form of a delay between the reading of a hardware clock and the time when this value is made available to the program.

The key measurements to be made in this area, then, are basic accuracy and effective precision[2] of the CLOCK function. Since it affects accuracy, the overhead incurred when reading the clock should also be measured. Other techniques for determining accuracy include comparing CLOCK readings with direct readings of hardware timers or virtual timers provided with the operating system. The effective precision of the CLOCK function is the resolution, or interval of time by which the clock is incremented at each tick. The CLOCK resolution can be determined using the second differencing technique described in [10] and discussed later in Chap. 3.

Knowledge of these timing parameters is essential in establishing a portable suite of benchmark programs. By identifying the accuracy and precision of the CLOCK function provided by the language, other benchmark programs may be run that use the CLOCK function for timing purposes and thus can be written in a machine independent manner. These programs are designed to compensate for coarse resolution and some inaccuracy in the CLOCK function, and they use the values obtained for accuracy and precision as parameters in their execution. This technique of benchmark program design is discussed at length in Chaps. 3 and 7 below.

2.1.1.2 Overhead and Latency

The other main area concerning the measurement of time in run-time benchmarks is that of execution overhead and latencies. Overhead is the time needed to execute an operation or set of operations that are defined at the language level. These operations are defined in terms of language features since different compilers will implement these operations in different ways. For example, the time needed to execute a subprogram call may be of interest, since it commonly involves basic operations like allocating a stack frame and altering the program counter. However, by using the subprogram call as the measured feature of interest in this case, an implementation independent measurement can be made that is easily expressed at the language level.

In addition to execution overhead, there are many types of latencies that can be measured. By latency, we mean the time that elapses between an event that occurs and the start of a second event that is dependent on the occurrence of the first. For example, there is a latency between the time when an exception is raised and the time when its handler begins execution. As is the case with execution overhead, there are a number of latencies that can be measured, and they are expressible at the language level.

With these classifications of execution speed measurements, we can identify a number of language characteristics to be benchmarked. For execution overhead measurements, the time to elaborate, activate, and terminate a task, allocate storage for program variables, perform a task rendezvous, execute various arithmetic operations, and evaluate run-time compiler attributes can all be considered in addition to the subprogram call mentioned above. In the case of latency measurements, interrupt response time, task execution after delay expiration, and exception handling latency are considered. In all of these cases, different parameters can be changed to determine their effect on the measured value. Of course, this is a partial list of measurements, but their classification yields a framework for creating new benchmark tests.

[2]Accuracy is the closeness of a measured value to the actual value of interest. It is determined by the amount of error in a measurement. Precision is the degree to which the measurement may be stated, i.e., the number of significant digits.

2.1.2 Space

The benchmarks developed in this area are intended to measure space utilization and capacity. The particular areas to consider in examining memory usage are the stack, the heap, and the code. The stack is a reserved area for storage where last-in first-out (LIFO) allocation is performed dynamically according to program structure and subprogram calling patterns. Because Ada is a multitasking language with support for block structuring and shared data, the stack is managed as a collection of stacks, one for each task. This organization is referred to as a *cactus stack* [29]. The heap is a pool of storage available for dynamic use by the program when the space required is used in a way that does not conform to a LIFO discipline or is too large to fit in the stack. The memory space needed for the code is determined by the compiler and should be read only. Additional considerations for space requirements are the effects of physical memory space and virtual memory space availability.

The aspects of the stack that are of interest for benchmarking include size, usage, and cactus stack structure. The size of the stack determines the amount of recursion possible and the threshold for changing stack allocations into heap allocations. Benchmarks can be designed to measure these values, and to gauge the effect of changing the task stack size via the attribute STORAGE_SIZE with a length clause. The usage of the stack may be affected by allocations that may be performed statically. Items that are normally allocated from the stack may instead be allocated from the heap or the base of the stack at compile time. This is common in the case of a **declare** block. Cactus stack organization is important because the way that the several task stacks are linked together will impact the performance of non-local memory references. The effect of this structure is best measured by determining the overhead of non-local references to task and block variables.

The structure and management of the heap can affect program efficiency in several ways. The overhead of heap allocation time is important, but it may be affected by factors such as the working size of the heap and the policy for increasing it. The interaction of both the physical and virtual memory spaces with the heap determine the overhead associated with increasing heap size.

A critical feature associated with heap size is the support for deallocation, either explicit or implicit. Explicit deallocation is by a call to a subprogram that relies on the run-time system to return the space to the heap. Implicit deallocation is supported by use of a garbage collector that is a process running concurrently with the program, detecting unused memory blocks and returning them to the heap. Both methods require some overhead, but their amounts may vary widely, and, in the case of garbage collection, may not be controllable by the programmer. It is also possible for a compiler to not support memory deallocation, thus reducing the amount of usable heap space over the lifetime of the program. Benchmarks can be easily designed to measure the deallocation policy and its impact on execution.

Measurements of code size for various program segments provide information regarding compiler code generation efficiency and compile-time versus run-time execution trade-offs. Effective use of registers, instruction reordering, and optimizations can reduce code size while providing for faster execution time. However, in some cases, code size may be increased by reserving space within the code segment. Furthermore, comparisons of code size across architectures will have little meaning, since instruction sets may vary widely.

2.1.3 Control

The amount of control over program execution available to the programmer is of great interest and can be measured in some cases. Although code ideally should be designed in a manner independent of the implementation's characteristics, in some cases this is not possible, particularly in real-time systems. Areas of program behavior where control is an issue include task delays, priorities, scheduling, entry call servicing, and run-time attribute evaluation. Knowledge of the implementation of task delays, timed entry calls, and select statements with delay alternatives is vital to a designer programming a real-time system. Biases in task scheduling as well as in support of priorities are also of interest to programmers requiring tight control on program execution. These biases as well as any in the servicing of entry calls grouped in a select statement may affect the way a programmer structures his or her code. Also, the speed and reliability of run-time attribute evaluation can impact a program's ability to react to dynamic changes in its environment.

The control issue with task delays is the accuracy and precision realized from a specified delay. High accuracy allows greater control over task execution intervals and desired execution in the cases of timed entry calls and accept statements. Precision in delays allows even greater control in these cases and also determines the minimum delay resolution. Benchmarks can be designed to determine minimum delay resolution and the effect of the delay implementation for varying delay values. A curve describing scheduler behavior in response to a delay can be determined as in [10]. Further tests involving timed entry calls and select statements with delay alternatives can be designed to assess their impact on delay precision and accuracy.

Biases in scheduling, priority support, and the servicing of entry calls affect the behavior of a program at synchronization points. If the behavior of the scheduler can be characterized, program execution may be more predictable. Benchmarks can be designed to determine which tasks become active and which are suspended at different scheduling points. These points can be brought about by task delays and task rendezvous. The impact of task priorities can be determined including their effect on scheduling overhead. The fairness of the servicing of a group of entries in a selective wait can also be determined by counting the calls accepted for each entry made by several tasks.

Run-time attribute evaluation is related to control because key program decisions can be based on this data. Examples of such attributes include TERMINATED for a task or COUNT for an entry, where TERMINATED is either true or false and COUNT is the number of calls queued on an entry. There are many possibilities for the run-time system to maintain this information either explicitly or implicitly depending on the system data structures used. It is important to the program that these queries be satisfied quickly and accurately. Benchmarks can measure the overhead of the call as well as determine how well these values are kept current.

2.1.4 Discussion

After examining the three main areas of performance, it is clear in the case of run-time benchmarking that many trade-offs between the three areas must be considered when designing the tests and interpreting the results. Such trade-offs include scheduling overhead versus fairness, storage allocation overhead versus heap management strategy, and space requirements for run-time system data structures and its related code size versus reliability of run-time attributes and fairness in entry call servicing. Also, there are trade-offs between compile-time and run-time operations that must be

considered. The areas for compile-time benchmarking are discussed below.

2.2 Compile-Time Benchmarks

Compile-time benchmarks can also be characterized using the dimensions of time, space, and control. While we can suggest different areas of compile-time performance that fall into these three categories, there is not yet a more formal description of compile-time benchmarks for the PIWG suite. The bulk of the benchmarking effort to date has concentrated on run-time performance. However, we can introduce this topic, which will aid in the design of effective benchmarks to measure compile-time speed, accuracy, and control. The following paragraphs are based on some issues that were raised at a PIWG workshop group meeting [22].

As stated above, different aspects of compile-time performance can be classified along the three dimensions of time, space, and control. The measurement of interest concerning time is, of course, compile speed, but this quantity is one that is difficult to measure precisely. There are many compile-time tradeoffs possible that can greatly affect a measurement of compile speed. For example, some compiler optimization and code generation may be deferred until link time on some systems, and these effects may escape a simple compile speed measurement. Another issue is the unit of measurement that should be used to measure compile speed. A measurement based on lines or semicolons compiled per second is not of much use if different programs are compiled for different systems. A common benchmark or set of benchmarks should be used to measure compile speed in all cases, and it should include the time for linking as an additional measurement. Also, consideration of compile-time/run-time composites should also be made, so that compilers with long compile times that produce short run times can still be viewed favorably. Additional measurements regarding compile speed should include the overhead incurred in initiating a compile job as well as the extra overhead induced by extensive optimization.

The compile-time space requirements of Ada compilers are measured in terms of several system resources. These resources include both temporary and permanent allocations of space in the memory hierarchy. At compile-time, a percentage of available main memory is allocated to the compiler as well as temporary disk space for virtual memory and files. Permanent disk space is required for libraries that aid in the management of Ada's separate compilation feature. These libraries are typically split between local libraries for a given program, and global libraries for all Ada programs. Some library components may be shared between programs, which in turn may reduce the permanent disk space requirement. In any case, some operating system dependent techniques will be required to measure a compiler's use of these space related resources.

Several control related measurements of compile-time performance are also possible. These measurements are mostly related to systems where multiple Ada projects are present or multiple users are cooperating on a single project. As mentioned above, the ability of multiple programs to share libraries is a feature that adds some amount of version consistency to multiple projects. Also, the ability of multiple users to compile different parts of the same program concurrently is one that enables more flexibility for the development team. Other control related measurements include the performance of the re-compilation system and any restrictions added to the compilation order semantics outlined in the LRM. Compiler systems including tools for compilation order management provide additional information to development teams and may prevent unnecessary re-compilation. In all cases, Ada modules will have to be developed to interact with the compiler's tool set to answer these questions.

Chapter 3

The Time Problem

Russell M. Clapp Trevor Mudge

The University of Michigan

The notion of time in Ada is necessary since the language is intended for use in programming embedded real-time systems. In this chapter, we examine time related issues in Ada and their relation to our approach to benchmarking. There are two aspects of the time problem area that must be considered in performance evaluation. The first is a need for benchmark programs to evaluate the timing related features of the language. Secondly, the existence of timing support provides the basis for developing a method within the language for measuring benchmark execution times. In order to develop an accurate and reliable method for timing measurement, the support for timing provided by the language must be calibrated using benchmark programs. We discuss these two inter-related areas of the time problem in the paragraphs below, after first reviewing the notion of time present in Ada.

3.1 Ada Time Units

There are several entities in Ada that relate to time. The following list of entities is reproduced from [10]:

- the data type TIME, objects of which type are used to hold an internal representation of an absolute point in time;
- the data type DURATION, objects of which type are used to hold values for intervals of time;
- the value DURATION'SMALL which gives an indication of the smallest interval of time which can be represented in a program. It is required to be less than or equal to 20 milliseconds, with a recommendation that it be as small as 50 microseconds;
- the value SYSTEM'TICK, which is defined as the basic system unit of time;
- a predefined package, CALENDAR, which provides functions to perform arithmetic on objects of type TIME or DURATION;

- a predefined function, CLOCK, which returns a value of type TIME corresponding to the current time;
- the operation **delay** which allows a task to suspend itself for a period of time.

The first two items are predefined data types for representing either points in time or intervals of time. A point in time has several components associated with it that represent various levels of precision in a time value, e.g., year, month, day, hour, minute, second, microsecond, etc. The type TIME is private, so the representation used should be hidden from the user. The multiple levels of precision suggest, though, that the representation of a point in time may be more complex than a single word value. Because of this, the overhead involved in computations involving values of type TIME will be different and probably greater than in the case of other predefined data types.

The representation for an interval of time is simpler. DURATION is a predefined fixed point type whose values are interpreted to be in seconds. Because it is a fixed point type, the values of objects of this type are multiples of the value DURATION'SMALL. This value, then, specifies the amount of the precision possible in representing an interval of time. It is important to note, however, that the value of DURATION'SMALL in no way implies performance of a system for time measurements or scheduling. The clock resolution is determined ultimately by the hardware. DURATION'SMALL is chosen based on word size and convenience of representing a wide range of interval values with a reasonable amount of precision.

A similar statement can be made regarding the timing environment available and the value SYSTEM.TICK. The phrase "basic system unit of time" is not very specific, and one might think it refers to one of several possibilities, e.g., time-slice interval, minimum delay value, CPU clock cycle, or hardware clock resolution. Several compilers do use this value to indicate the resolution of the CLOCK function, but this is not always the case. It is best to determine this resolution experimentally, as described in later sections.

The CALENDAR package contains subprograms to perform arithmetic on objects of types TIME and DURATION as well as extract date components from an object of type TIME or combine such components into a single TIME object. The CALENDAR package also contains the CLOCK function. The CLOCK function returns a value based on some underlying hardware clock or timer. The increment in time to the clock or timer is the resolution time of the system. If this increment is the same as the tick for the CLOCK function, it is also the CLOCK resolution. If the execution overhead for evaluating the CLOCK function is less than its resolution, then successive calls to the CLOCK function may return the same result.

The **delay** operation allows scheduling and synchronization of tasks to be influenced by time. Delays are specified with a time interval of type DURATION. A delay may suspend a task for an amount of time no less than that specified, or the delay may be used in conjunction with an entry call or one or more accept statements. In the case of rendezvous related delays, the delay specifies a bound on the time a calling or accepting task is allowed to wait for a specified rendezvous to begin. In any case, there are several possibilities for the implementation of the delay operation. The LRM places no limits on the length of a delay, except that it last at least as long as the time interval specified. This situation gives an implementation a lot of freedom, and the particular implementation strategy used can greatly affect a program's execution.

3.2 Timing Techniques

There are many factors which come into play when determining the method to be used for timing overhead and latencies in benchmark programs. The most important issue to address is the structuring of the benchmark code. It is necessary to ensure that the benchmark measures exactly what it is designed to measure. It is also important that the timing environment be well understood, so that the desired accuracy and precision in the measurement can be obtained. Additionally, careful examination of the results must be made, so that undesired effects from the hardware, operating system, or run-time system can be discounted. We discuss each of these areas in detail below.

3.2.1 Feature Isolation

In order to measure a latency time or overhead for a specific language feature or set of statements, it is necessary to subtract out any overhead for control statements that aid in performing the measurement. In order to do this, the overhead of the control statements must be measured separately. This is done by textually repeating the code for the test but omitting the feature or set of statements that are the target of the benchmark. Because the resolution of the timer is usually greater than the time being measured, it is common for the benchmark code to include loops around the features being measured so that they are repeated a large number of times. The measured value is then divided by the number of iterations. The form of the overall benchmark, then, is made up of two loops: the test loop and the control loop. This benchmarking technique has been referred to as the "dual loop" approach [3].

Although this technique was designed for measuring language feature overhead, it can also be used to measure running times for synthetic benchmarks and applications. The feature statement is simply replaced by the statements that make up the application being measured. This technique may not be necessary if the running time of the synthetic benchmark or application is long in comparison to the CLOCK resolution. If this is not the case, the dual loop approach provides a way to obtain more accuracy and precision in the measurement through the use of repetition with timed feature isolation.

In order for dual loop coding approach to be successful, it is important to ensure that the measurements it produces are valid. The major factor to consider when designing the dual loop code is the effect of compiler optimizations. Although code optimizations are generally welcomed as a way to improve performance, they can also invalidate a measurement made by using the dual loop approach. Techniques must be incorporated to prevent an optimizing compiler from changing the program in any way that invalidates a measurement. The code generated for the test loop and control loop must not be different, except for the quantity being measured. Of course, any optimizations to the actual feature being measured welcomed and and encouraged. Chapter 6 discusses possible compiler optimizations. Chapter 7 provides examples of the dual loop code structure used in the PIWG benchmarking suite.

3.2.2 Timing Accuracy and Precision

In addition to correctly structuring the code for the benchmark tests, it is important to evaluate the timing mechanisms available and use them properly. The accuracy and precision of any clocks used must be measured, since these values determine the running time of the benchmark, which in turn determines the precision and accuracy of the results due to the measurement technique.

3.2.2.1 Clocks

There are several options possible when choosing a timing mechanism for use in benchmark programs. The most obvious choice in this case is the Ada CLOCK function of the CALENDAR package. The CLOCK function measures real time (also referred to as "wall clock time") by definition of the language. Using the CLOCK function for timing in benchmarks has both advantages and disadvantages. The main advantage is portability, since the time intervals may be determined by taking the difference of two CLOCK readings in an implementation independent manner. The main disadvantage is that, since CLOCK returns real time, the benchmark must be run in a situation where it has sole access to all available CPU cycles. If any intervening processes are present, e.g., in a time-shared system, they will steal away CPU cycles from the benchmark while the CLOCK is still ticking. This situation would provide an inaccurate measure of the benchmark's running time.

Another possibility for timing in benchmark programs is to use a CPU or "virtual" timer. A timer of this type measures the time actually used by the program. It does not tick when the program is blocked and another job is executing. This type of timer is generally provided in time-shared systems (e.g. Unix and VMS). There are two main disadvantages in using CPU timers. The first is portability. Although the timer may be invoked by calling a language level function, that function must be rewritten for each operating system that the benchmark programs are run with. The second disadvantage is the degree of uncertainty present when using CPU timers. In the presence of other jobs, CPU timers charge ticks to the running process when the wall clock is updated. Since context switches can occur at any time, it is possible for time to be charged to the active processes inaccurately. A few informal tests have been done at Michigan that demonstrates variability of CPU time readings in the presence of additional system load. While the timers may be accurate enough in the long run for accounting purposes, a confidence level for short time periods must be established before using them in conjunction with benchmark programs.

Yet another option for timing in benchmark programs is to use machine dependent hardware timers. These timers can generally be read directly without operating system interference, and they usually measure real time. When they are present, hardware timers usually provide a finer resolution than timers available through the operating system. These timers are critical in the case of embedded systems where there is no operating system. While lower reading overhead and higher resolution are desirable qualities for a timer, the problem of portability still remains in this case.

In the case of embedded systems, clock resolution and accuracy are even more important. In order to maximize these features in such a system, logic analyzers and other high resolution external hardware timers can be used. While providing the highest amount of accuracy and precision possible, these timers often require knowledge of the assembly code generated by the benchmark so that a trigger for the external timer can be determined. This type of timing is not at all machine independent, and cannot be expressed at the language level.

The PIWG suite currently uses both the Ada CLOCK function and an operating system dependent CPU timer for measuring time intervals. Because the CLOCK function measures real time and the CPU time may vary with system load, it is required that the benchmark program be the only active program in the system. When available or appropriate, the definition of the system dependent clock function can be changed to take advantage of additional accuracy and precision, as described above. The use of multiple clocks provides some additional confidence in the results when they agree, and can help pinpoint problems when they do not.

3.2.2.2 Clock Resolution

Independent of which type of clock is used to measure time intervals, its precision must be determined since this affects the benchmark's running time and the precision of the results. As stated in Chap. 2, the effective precision of a clock is the resolution, or interval of time by which the clock is incremented at each tick. This value can be determined using the second differencing technique explained in detail in [11]. We outline the basic approach below.

The idea behind second differencing is that, independent of the time needed to read the clock and its effective precision, the resolution can be determined by a series of nearly consecutive, equally spaced clock readings. By equally spaced, we mean clock readings with the same set of other instructions being executed between each read. If we assume that the resolution can be exactly represented by a model number that is an integer multiple of DURATION'SMALL, then we can determine the clock resolution by taking the second difference of the string of values produced from the consecutive clock readings.

A sequence of first differences is produced by computing the difference between all adjacent values in the original sequence of clock readings. Applying this differencing operation again to the sequence of first differences produces a sequence of second differences. As stated in [10], the values present in the second difference sequence are one of τ, $-\tau$, or 0, where τ is the clock resolution. Furthermore, as stated in [10], the number of zeros present in the second difference sequence can be bounded and reduced by adding instructions between consecutive readings of the clock.

If we drop the assumption that the resolution may be exactly represented by a model number for type DURATION, the process of determining the clock resolution becomes more involved. Depending on the implementation of the CLOCK function or some other clock routine and its Ada interface, it is possible for first or second difference values to be equal to DURATION'SMALL when they should be equal to 0. In this situation, the larger of any values present in a second difference string should be regarded as the clock resolution. This problem arises because a value that should be equal to the clock resolution can instead only approximate it. The difference between this value and the actual clock resolution is less than DURATION'SMALL. Thus, two distinct values that differ by DURATION'SMALL could both be representations of the clock resolution. This problem is addressed in more detail in the paper by Pollack and Campbell that appears in Chap. 7.

Another problem that has occurred with this technique of clock calibration is the case where the real time clock of a system is artificially incremented by a value much smaller than the resolution between consecutive clock readings. This change in the value of the clock is not due to a tick, but instead occurs because the operating system wants to guarantee that no two clock readings will return the same result. This property is useful when there is a need to dynamically generate unique identifiers, but it confuses the semantics of the clock.

This style of real time clock implementation has been observed at Michigan and by others [36] in the Sun-3 and Sun-4 series computers. The clock is incremented by one microsecond between readings unless enough time has passed to tick the clock to the next even multiple of 20 milliseconds. When calibrating clocks of this type, closer scrutiny is required when examining the original readings, the first difference sequence, and second difference sequences.

3.2.2.3 Accuracy

Clock accuracy is important to gauge as it determines the degree of reliability of benchmark results. It is also desirable to measure the accuracy of the CLOCK function since it is a performance parameter of the compiler and run-time system. One way to measure accuracy is to compare clock readings with values from another clock. At the language and system level, this would involve comparing multiple consecutive readings of all clocks available, e.g., CLOCK, a CPU clock, a hardware clock, or an external timer. Since all of the internal clocks may rely on the same hardware timer, we would expect them to agree for the most part. Any discrepancies may expose differences in the implementation or varying amounts of overhead in reading the clock. Ultimately, in order to verify the validity of the timing hardware, the updating of the clock value must be compared against an external standard.

Another issue that is important to the real-time programmer and which may affect accuracy is the overhead encountered in reading the clock. This is mainly a concern of the CLOCK function provided by the language. Measuring overhead and comparing it to the resolution will give an indication of how "stale" the CLOCK reading may be. Also, if a programmer wishes to use the CLOCK function as part of a calculation to determine a value for a **delay** statement, the overhead involved in calling CLOCK could be critical.

However, the overhead involved in reading the clock does not theoretically affect the second differencing technique described above. A longer overhead has an effect analogous to inserting other instructions between clock readings. If the clock overhead is greater than the resolution, though, it is necessary to compute the second differences to determine the clock resolution. If the overhead is less than the resolution, computing the first differences is sufficient to produce the resolution.

As for measuring time intervals for benchmarks, the clock overhead will not greatly affect the results as long as it is constant. Intuitively, this results from the notion that the overhead will be split into two pieces, the part before the actual instruction that reads a time register and the part that occurs after. Two readings of the clock then measure the running time of the instructions executing between the calls as well as one after part and one before part of the clock reading overhead. Therefore, a timed interval contains the total time needed for one reading of the clock. If the overhead is on the order of the clock resolution, its effect on the measurement will be diminished by the number of times the measured feature was repeated. If the overhead is much greater than the resolution, its value can be subtracted from each time interval measured.

3.2.2.4 Measurement Precision and Accuracy

The precision of the time measurements is determined by the resolution of the clock used and the number of times the feature being measured in the loop was repeated. Because loops are used, we call this repetition count the number of *iterations*. As reported in [10], the time a loop takes to execute and the time measured for it using a clock can differ by as much as the resolution of the clock. When this value is divided by the number of iterations (call it N), we find that the measured value for a single execution of a feature is within a value τ/N of the actual time (where τ is the clock resolution). Assuming no other effects are present that invalidate the measurement, we can say that the measurement is to a precision of τ/N, and the accuracy of the reported value is within $\pm\tau/N$ of the actual value. Since τ/N indicates a bound on the amount that the measured value can vary

from the actual, a percentage of certainty can be calculated. If it is desired that the reported results be within 1% of the actual, one only need guarantee that τ/N be less than 1% of the measured time. This can be done by increasing the number of iterations until the condition is met.

3.2.2.5 Measurement Stability

Because of several different effects that interfere with benchmark measurements (as described below), we would like to specify some stability criteria to help increase confidence in a measurement. Since the clock resolution can be determined and the number of iterations is under programmer control, the amount of error introduced by the repetition technique alone can be minimized. As stated above, we can limit this particular source of error to 1% or even less.

To reach this goal of 1% error, the number of iterations for a test is increased until the difference in times for the test loop and the control loop is greater than $100 * \tau$. If we assume no other effects are present that may skew the results, the value determined for the measurement will be within one clock tick of the actual time used. $\pm\tau$ compared to $100 * \tau$ is an error of $\pm$ 1%. The value reported for a single iteration is also within 1%, the precision of this value is τ/N as stated above.

In situations where the clock resolution is coarse and the feature being measured requires very little time to execute, the number of iterations may be prohibitively large. Allowances for additional error must be made in these cases and considered when analyzing the results. Also, it is important to note that other effects may be present that introduce additional error into the results. These effects are described in the next section.

3.2.3 Problems with Dual Loop Approach

There are several problems that can occur in different compiler implementations running on different computer systems that can distort the benchmark results when using the measurement techniques described above. We classify the major problems into three categories:

1. Translation Anomalies
2. Code Placement Effects
3. Operating and Run-Time System Interference

Although 3. was discussed in [10], points 1. and 2. have been identified by users of the PIWG and Michigan suites since 1986. All three of these problems have been discussed in detail in [2]. They were also discussed in April 1988 at a PIWG workshop [34].

3.2.3.1 Translation Anomalies

A translation anomaly occurs when the compiler generates correct code that differs from that expected. The principle causes of translation anomalies result from a failure in the technique used for blocking an optimizer or from a compiler that generates different machine code for two pieces of textually identical Ada code. The first case occurs when improper techniques for optimization blocking are used.

A concerted effort must be made at design time to ensure that all optimization blocking techniques cannot be circumvented. Prototyping the benchmark on several systems can also aid in detecting loopholes that optimizers might find, but this should not be used exclusively.

The second case of translation anomaly can occur if a compiler generates machine code based on some global conditions in addition to the original Ada source. For example, a subprogram's entry code may be longer or shorter if it is the first one declared in a package. Other examples include subprogram calls being long or short based on their relative position to the caller or NOP instructions being added in some cases to provide word alignment. These problems can be detected by repeating the benchmark loops several time textually and measuring times for all the control and test loops separately. If more information is known about why any discrepancies may exist (e.g. compiler generated assembly code is available), some determination may be made as to which loop times should be considered valid. This code repetition technique can also detect other anomalies as we shall see next.

3.2.3.2 Code Placement Effects

Additional timing anomalies can occur when running benchmarks due to the effect of code placement in memory. Experiments have shown that textually identical loops may have different running times if code for a loop crosses a page boundary (even if there is no page fault) or is aligned on a halfword boundary [2].

Other problems arise if the benchmark code cannot entirely fit inside a cache (if present) or inside of main memory. Cache misses and/or page faults invoke system activity and increase execution overhead. If these effects are not equally spread across all timing loops, the results will be distorted. In [10], the recommendation was made that all benchmarks be made small enough to avoid the effects of paging. In some cases, this requires a benchmark to "undo" certain actions previously done, so that large amounts of memory are not consumed [7]. When caches are present, it would be preferable if the benchmark code could reside entirely in the cache. A run through the code before timing begins can bring the code into (or "warm up") the cache and main memory.

Testing with multiple textually identical loops can aid in detecting any of these conditions. The benchmarks should also be repeated textually within the code with multiple complete runs made to determine if any of these conditions are significantly affecting the results.

3.2.3.3 Operating and Run-Time System Interference

Interference from the operating system and/or run-time system can also invalidate benchmark results. As mentioned in [10], it is possible for an operating system to time-slice a benchmark and check for other processes to run even if the benchmark is the only process in the system. It is also possible for system daemon processes to become active and compete with the benchmark for CPU time. The run-time system can interfere with the benchmark as well if, for example, it begins a background task to perform garbage collection.

Many of these effects can be avoided if system daemons can be disabled. If some problems still occur, wild distortions appearing in the results can be discarded if they are very infrequent. Some effects cannot be avoided, but will probably involve only a small cost. When running the Michigan

benchmarks under Unix, it was estimated that operating system interference consumed no more than 5% of the run time [10]. The Michigan approach was to repeat the tests several times and assume that the smallest running times for both the control and test loops represented the cases where system interference was minimal. The PIWG approach involves an average of running times with extreme values on both ends being discarded. This will tend to average the system costs into the results. Other approaches to measuring this effect or factoring it out are possible.

Chapter 4

The Space Problem

Russell M. Clapp Trevor Mudge

The University of Michigan

The concept of space used by a program comes from the fact that programs process data and the notion that a computer's memory acts as a store for that data. Because this need for space by a program is so basic, it is an important area for performance measurement. In this chapter, we identify the various abstract forms that represent memory usage in an Ada program. We examine these forms as they appear at the language level as well as in the run-time system. In addition, various types of space usage measurements are discussed as well as techniques for performing them.

4.1 Ada Storage Units

There are several different forms of storage representation in an Ada program. We classify these into two major areas: 1) program-level storage, and 2) run-time system storage. Program-level storage are those constants and variables declared by a programmer. Run-time system storage involves the data structures used to support the execution of the program and the code space. Instances of both types of storage are discussed below.

4.1.1 Program-Level Storage

There are several different notions of storage available at the language level in Ada. There are several predefined types as well as the capability to construct user defined types. These data types can be classified as either scalar or composite types. Scalars include INTEGER, FLOAT, fixed, and enumeration type variables while composites include record and array types. Composite types may be made up of values that are also of a composite type, e.g., an array of records. Another object that can be declared in a program is a task. Tasks can be considered composite objects which are also executable.

All of these different types of program-level objects can be declared both statically and dynamically. Dynamic allocation can occur on entry to a subprogram or **declare** block where the object dimensions are known either statically or at run-time. Dynamic allocation can also be explicit using the **new** allocator, which can be used to create a single object (potentially composite) of a particular type. As an optimization, many objects can be allocated at compile time. In addition to package variables, this optimization is commonly used with **declare** blocks, even when some objects are of unknown dimensions. In this case, the maximum possible size is allocated. This is clearly a space/time tradeoff.

Storage allocation strategies may also be impacted by the **pragma** PACK and the attributes 'STORAGE_SIZE and 'SIZE. The PACK **pragma** can be used to tell the compiler to use space minimization as the criterion when selecting a format for the representation of the specified array or record type. While possibly reducing the size of objects of the PACKed type, this **pragma** may also affect the overhead involved in manipulating objects of this type. The attributes 'STORAGE_SIZE and 'SIZE on the other hand can be used with representation clauses to direct the compiler to use specified sizes for object representation and storage availability. Changing sizes with these attributes may also affect run-time overhead in allocating and manipulating these objects. This is discussed further below.

4.1.2 Run-Time System Storage

Although program-level storage structures are clearly defined, the structures used by the run-time system are implementation dependent. Still, there are some indications as to the types of structures needed by the run-time system given in the Rationale [29], the Language Reference Manual [17], and traditional implementation style of procedural languages.

Two main structures we recognize as part of the run-time system are the cactus stack and the heap. The cactus stack is the program stack which is spread across all tasks of the program. The "trunk" of the stack is that part owned by a parent task and shared by all its descendent tasks. The portions of the stack that belong to a particular task are managed as a typical run-time stack for that task. The heap space is provided to the program as a storage area for dynamically sized and allocated objects.

Another structure the run-time system may require are task control blocks (TCB's). These control blocks hold information and pointers required to manage a task's execution. TCB's can then be used as entries for entry queues, a run queue, a blocked task queue, or other queues. The types of queues used and their management is implementation dependent.

4.2 Usage Measurement

There are three main areas for measurement regarding space: usage, capacity, and reclaimation. Examples of usage measurements include determination of how much space is required to support language level entities such as INTEGER scalars and tasks. Determination of the capacity of both the heap and the cactus stack is also of interest. Detection of either explicit or implicit support of storage reclaimation is also important. In addition to this measurement of space, we would like to measure

the time overheads involved in space manipulating operations. We will consider tests of this type to fall under the domain of time consuming feature tests.

4.2.1 Usage

The tests needed in this area are ones that determine the amount of memory needed to represent different language-level structures. This can be accomplished by examining the values of SYSTEM.STORAGE_UNIT and the attributes 'SIZE and 'STORAGE_SIZE. The minimum number of bits required to represent objects of any type can be determined by directly reading the attribute 'SIZE for these types. The actual number of bits used to represent an object already allocated can be determined by reading the attribute 'SIZE for that object. A reading of the attribute 'STORAGE_SIZE for a type yields the number of storage units reserved for the collection of objects of that type. In the case of a task type, however, 'STORAGE_SIZE yields the number of storage units reserved for an activation of a task of that type. SYSTEM.STORAGE_UNIT is a constant equal to the number of bits per storage unit, and is an implementation dependent quantity.

Another measurement of some interest is code size. Although the values here can vary widely from architecture to architecture, the data here can be interesting when comparing different compilers or even different languages on the same architecture.

4.2.2 Capacity

The test needed in this area are ones intended to determine the capacity of the cactus stack and the heap. For a local stack segment, capacity can be determined by measuring the number of recursive calls a procedures may make. This technique, however, requires knowledge of how much stack space a recursive procedure call requires for a particular procedure. The more local data the procedure allocates, the fewer calls that are necessary to raise the exception STORAGE_ERROR. This method may be even trickier if stack expansion techniques involving the heap are employed by the compiler to avoid the exception. Experimentation in this area may uncover other useful measurements that can be made.

The capacity of the heap can be determined by making many dynamic allocations with the **new** allocator and keeping references to the data. A count of dynamically allocated arrays of INTEGERs can be kept, and, if the size of the INTEGER array is known to or set by the program (through 'SIZE), the capacity of the heap can be determined when STORAGE_ERROR is raised. The handler for the exception can check the count of allocated arrays. Pointers to the arrays are kept in arrays of access type objects. These references are necessary to keep any automatic form of space reclaimation from increasing the heap size. An implementation of this type of test is described in [10]. However, difficulties may arise if storage space available for arrays of INTEGER components is less than the total heap size. Consideration of operating system limits, the values of 'STORAGE_SIZE for different types, and the results of the allocation test just described must be combined to determine the performance of the heap.

4.2.3 Reclaimation

The heap allocation test described above can be used with minor changes to detect the presence and performance of storage reclaimation techniques, as described in [10]. If references to the allocated arrays are overwritten with references to newly allocated data, the data no longer accessible is implicitly freed. If some form of garbage collection is present, more arrays can be allocated before STORAGE_ERROR is raised or the exception may not occur at all. If clock readings are saved after every allocation as suggested in [10], some conclusions about the run-time overhead involved can be made. Alternatively, space can be explicitly deallocated using an instantiation of the predefined generic UNCHECKED_DEALLOCATION. If space is reclaimed by this subprogram, STORAGE_ERROR should not be raised at all. Clock readings for this test may provide an interesting comparison to the ones available from the other two forms of this allocation test.

4.2.4 Overhead

As stated above, these tests fall under the domain of feature overhead tests. Areas where storage related overhead measurements should be made include allocation with both static and dynamic bounds as well as with the **new** allocator. Tests of this type are described in [10]. Other overhead related tests include access and modification times for objects whose representations are affected by the **pragma** PACK and representation clauses involving 'STORAGE_SIZE and 'SIZE. These tests can be designed using the techniques described in Chaps. 3 and 6. An additional concern for optimization blocking here will be the prevention of static data allocation. This can be accomplished by allocating space within a subprogram.

Chapter 5

Parallel and Distributed Issues

Russell M. Clapp Trevor Mudge

The University of Michigan

The designers of Ada apparently had in mind the notion that Ada programs could be written to execute on parallel and distributed systems [17]. Further, it is clear that recent trends in computer hardware will make such systems commonplace, and for some embedded applications, the most appropriate. In this chapter we will discuss some of the performance issues pertinent to parallel and distributed systems. However, there are very few Ada compilers for parallel or distributed systems and even the notion of a distributed Ada program is ill-defined [43]. Bearing this in mind then, our comments in this chapter should be considered much more speculative than those in the other chapters.

Parallel and distributed systems can conveniently be divided into two classes: those in which program units are assigned processors at run-time, and those in which program units are assigned processors at or before compile time. For the purposes of this discussion we will term the former parallel systems and the latter distributed systems. Of course, realistic systems are likely to be mixtures of the two. Parallel systems are typified by shared-memory multiprocessors in which no more than a dozen or so processors share a common memory. This style of machine architecture has had notable commercial success; examples are the Sequent, Encore, and Alliant series of multiprocessors, all of which have parallel Ada compilers available. Distributed systems are typified by a system that has loosely coupled (distributed memory) processors dedicated to specific functions. For example, microprocessors may be placed at the site of sensors to perform on-the-spot data conditioning or compression and connected to a coordinating processor by a LAN/SAN or simply a twisted pair.

In order to take system-wide advantage of strong typing, packages, tasking, exceptions and other features of Ada, both parallel and distributed systems should be programmed as a single Ada program [12]. Given the need to do this, what is the appropriate "unit of distribution" ?

5.1 Distributed Systems

Within the current definition of Ada, it is not possible to define a unit of distribution that is without some shortcomings for distributed systems [43]. Our recommendation is that library subprograms and library packages be the only distributable units [35]. They represent a reasonable level of granularity for distribution, and they provide a reasonable unit for structuring distributed programs. Furthermore, as shown in [43], they do not require cross-machine dynamic scope management. This fact greatly simplifies compiler implementation and run-time support, allowing efficient cross-machine communications. Additional recommendations are:

1. Data objects created from remotely defined types should be placed with the unit creating them, with implicit and basic operations being replicated. User defined operations should remain on the unit elaborating the corresponding type definition.

2. Task objects should be placed with the unit initiating their creation.

The above recommendations could be presented as guidelines for the programming of distributed systems without requiring changes in the language definition. However, compiler and run-time support would still be needed for those situations that may arise should the user decide to ignore the guidelines. To avoid this, we have advocated that the recommendations should be part of the language definition [35, 43]. They represent a compromise that will require a minimum of reinterpretation and augmentation to the present language definition, while greatly facilitating the programming of distributed systems.

A more radical departure from the current language, and one that we would also like to see, is to have package types added to the language specification [9], [30]. This would allow the further recommendation that task objects created from task type definitions be restricted to the units where the corresponding type definitions are elaborated. This would simplify distributed task termination without restricting tasks of the same type to the same processor. Package types would also be compatible with the programming of parallel systems, simplifying the programming of systems that had elements of both parallel and distributed systems.

However, other approaches to program distribution are possible, as noted in [31, 32]. The approach suggested there involves the use of a distribution specification written in the Ada Program Partitioning Language (APPL). This allows an existing program to be distributed across nodes based on the specification. the current implementation described in [31] is being built to allow distribution of library packages, library subprograms, and objects, tasks, and subprograms that are declared in the visible part of a library package. This approach allows more distributable units than the approach suggested in [43]. It is similar to the approach suggested in [12, 13], but allows objects and subprograms to be distributable in addition to tasks that are visible in library package specifications.

In the longer term, however, the implementation described in [31] is intended to support the finer grained distribution of all objects, subprograms, and tasks. As stated in the reference, this will require more support from the run-time system. The motivation for this approach is based on the flexibility it gives the programmer and the ease of reconfiguration made possible by APPL. If a distribution is selected that requires an excessive amount of run-time system overhead, it may be reconfigured by changing the APPL specification so that a more reasonable overhead may result.

Of course, there are many other approaches possible for distributing Ada programs. These include the addition of language level interface code between processing elements [5] and using entire programs as the unit of distribution (described in [15]). The issues raised regarding performance in distributed systems apply to any distribution strategy.

5.2 Parallel Systems

The natural unit of distribution for execution targets that are shared-memory multiprocessors is the task. Indeed, the shared-memory multiprocessor (in the limiting case a single processor) appears to be the model of computation that the designers of Ada had in mind. For example, Ada's block structured scoping rules seem to imply a shared memory model. When shared memory is present, and its access is transparent to the user program, the difficulties discussed in the previous section are eliminated. Because memory is shared among all processors, only active units of execution (i.e. tasks) need be allocated to processors. They can be scheduled at run-time based on available processors, or, if their number is known statically, tasks can be bound to processors at compile, link, or load time [14]. Combinations of these scheduling approaches are also possible, including the option of binding of a subset of tasks to a certain processor.

The notion of binding objects, packages, and subprograms to processors is unnecessary in a transparent shared-memory multiprocessor. These passive objects simply reside in memory and are directly accessible by all tasks to which they are visible [1]. In the case of subprograms, execution of the object is also involved. However, as long as the code is reentrant (which it should be), all subprogram calls can be executed locally by any processor in the system by simply sharing one copy of the code. This creates a possibility of multiple simultaneous executions of a single subprogram. This should be a desirable situation, because it increases parallelism. If a resource needs protection from simultaneous execution or access, it should be guarded by a task and synchronized through the use of rendezvous. As stated above, tasks can be bound to processors or allowed to freely migrate as the needs of the application code dictate.

Non-uniform memory access (NUMA) machines can be considered differently. Although most machines of this type provide a globally accessible shared address space (e.g. BBN Butterfly and IBM RP3), there is the issue of performance which can be determined by the placement of objects in memory. These types of machines typically provide a local memory for each processor along with a global memory or a method of direct access to other processors' local memory. In this situation, it may be desirable or necessary to distribute and/or replicate subprograms across local memories to improve performance. Care must also be taken in distributing objects to local memories or keeping them in a global memory. In the RP3, the local memory acts as a cache that other processors cannot access. Shared data must be non-cacheable in machines of this type, or either a hardware or software implemented consistency scheme must be employed [38]. Clearly, the location of shared objects in the global address space provided by NUMA machines is a critical factor in determining program performance and correctness.

[1] This is true even when each processors has its own private cache, provided that cache consistency is maintained in hardware or by the run-time system and the caches are transparent to the user program [14].

5.3 Performance Issues

Because of the wide variety of implementation strategies available to implementors of distributed and parallel Ada systems, there are a large number of performance measurements to be made. In addition to the performance benchmarks run on uniprocessor systems, benchmarks must also be run to gauge the impact of distribution and parallelism. However, there are many problems to be overcome in developing benchmarks for this area, and many will not be solved, or even identified, until there are complete implementations available for experimentation. We discuss some of the more obvious of these problems and approaches in the subsections below, recognizing that the discussion can only be regarded as speculative at this stage in the development of distributed and parallel systems.

5.3.1 Language Feature Tests

When measuring the overhead, latency, capacity, control, etc. associated with various language features, significant impact may result from distribution and parallelism issues. Particularly, additional run-time system support may be necessary to support language features in some cases, and this may reflect negatively on performance (e.g. remote vs. local subprogram call). In fact, there may be many more features where performance measurement is vital in the parallel and distributed cases, that seem unimportant in the uniprocessor case, e.g., object reference overhead. Distribution strategy and scheduling techniques can further complicate this problem by determining which items are remote [14]. Alternatively, some benchmarks for language features may indicate improved performance for parallel and distributed systems over uniprocessor systems. For example, the time needed to elaborate, activate, and terminate a large number of tasks in a parallel system may be much less than for a uniprocessor system. The question arises then as to what exactly the benchmark is intended to measure. As discussed in Chap. 3, looping is often used to repeat a measurement so that the coarse clock resolution may be compensated for and a meaningful result produced for a single execution. If the execution of such a loop is shared by multiple processors, the time to perform one iteration is no longer easily determinable. Optimization blocking and program structuring techniques may prevent this situation in some cases, but the question remains as to whether that should be the intent of the performance measurement. If repeated operations can be speeded through parallel computation, then this is something we would like to identify and measure.

5.3.2 Synthetic and Application Benchmarks

In this area of benchmarking, we would hope to see an improvement in performance on parallel and distributed systems. As before, however, there are many issues to consider. While multiple processing elements should, in principle, contribute to the speedup of an application or synthetic benchmark, there is some question as to whether or not the program is distributable or parallelizable. This is determined by the units of distribution for such systems. While this potential speedup may be largely dependent on the number of executable tasks, there is also a possibility in parallel systems of vectorizing and parallelizing optimizations for loops and multiple procedure calls. The benchmarker must decide whether or not to restructure the code to either aid the compiler in making these optimizations or increase the amount of distributable tasking possible. While such a restructured program may run slower in the uniprocessor case, it will hopefully run much faster in the multiprocessing case than the best sequential algorithm.

Clearly, effects of program design, code restructuring, and distribution specification can have a great impact on the results produced for parallel and distributed systems. Guidelines for performing these measurements must be developed as the issues are uncovered.

5.3.3 Compile-Time Benchmarks

The performance of compile-time benchmarks may also be greatly affected in parallel and distributed systems. When multiple compiles are necessary for a particular program, it may be possible to perform them in parallel, reducing compile time. Also, in some cases, the front-end of a compiler implementation may be parallelized, allowing for speedup of individual compiles. On the other hand, linking and loading may be more complex for distributed and parallel systems, requiring more time and space than the uniprocessor version. These considerations must be fully explored, so that the extent of multiprocessing at compile time can be quantified and its advantages and disadvantages be understood.

5.3.4 Benchmarking Environment

In addition to the concerns raised regarding the benchmark results and their interpretation on multiprocessing systems, we must also examine the benchmarking environment presented by these systems. These environments may preclude any meaningful results from being generated from our existing collection of benchmark programs. Measurement of time and space consumption may be very difficult in the presence of multiple address spaces and multiple timers. It is difficult to predict the type of timing environment provided by a distributed or a parallel Ada system. Several possibilities exist [42], and each one affects clock overhead and clock resolution differently. Benchmarks designed to evaluate this environment may have to be redesigned for different parallel and distributed systems. Techniques may also be necessary for aggregating space and time usage among multiple memory and processing elements to determine a program total.

5.4 Summary

Clearly, there are many issues that must be addressed before the benchmark suite can be considered appropriate for parallel and distributed systems. These issues include the benchmarking environment provided, the structure and content of the benchmark programs, and the interpretation of the results they provide.

Chapter 6

Optimization

Russell M. Clapp Trevor Mudge

The University of Michigan

Compiler optimization is an important issue in Ada performance evaluation for two major reasons. First, compile-time optimization techniques for Ada compilers must be understood in order to develop benchmark programs that correctly measure the performance of a language feature. As we have stated previously, if benchmark programs are not structured carefully, compile-time optimization can modify the code in a way that distorts the intended measurement. Second, in order to completely evaluate the capability and performance of an Ada compiler, tests are also needed to measure a compilers ability to perform certain compile-time optimizations. In tests such as these, our aim is to determine the presence, degree, or absence of certain optimization techniques in an Ada compiler. An example of such a test would be the performance measurement of a compute-intensive synthetic benchmark that was compiler with and without full optimization. These sorts of tests will appear more often in future benchmarking suites.

The need to defeat optimization was introduced in Chap. 3, and the techniques used in the PIWG suite are discussed in Chap. 7. Again, we emphasize that this need to defeat optimization is intended to ensure the accuracy of the intended benchmark measurement. It is important that the compiler does not reduce the repetition introduced with the dual loop approach, or eliminate entirely the control loops that are intended to measure looping overhead without computing any other useful result. On the other hand, any optimization techniques that can improve the performance of the feature being measured are welcomed, as they demonstrate the performance capabilities of the code generated by the compiler.

In order to correctly block certain types of optimization as well as detect the ability of the compiler to perform other optimizations, an understanding of possible Ada compiler optimizations is needed. The following paper discusses the problems and possibilities of compile-time optimization for Ada, and helps to provide the background needed to develop correct benchmark programs.

Chapter 8

Recommendations and Future Trends

Russell M. Clapp Trevor Mudge

The University of Michigan

Daniel Roy

Ford Aerospace

8.1 Run-Time Parameters

As should be clear from the previous chapters of this document, precisely identifying and quantifying all run-time parameters is a very difficult task, and, in some cases, is impossible. Between all of the different types of benchmark interference and the great latitude given to compiler implementors, several areas of performance measurement are extremely difficult. Determining the run-time system's scheduling algorithm is an example of a very difficult measurement [10].

We would like to recommend that vendors supply more information regarding their run-time environments in a manner that can be incorporated into the benchmarking effort. PIWG should determine a list of critical features of a compiler and run-time system implementation that should be provided by implementors. Then, benchmarks can be developed to verify the information provided. We believe that verification of the information and performance measurement of these mechanisms is an easier task than attempting to develop programs that determine these parameters. PIWG's initial attempts to address this problem are described in the results part of this special issue.

8.2 Future Trends

As explained in Chap. 2, run-time benchmarks can be characterized by the dimensions of time, space and control.

The current PIWG suite only addresses the time dimension. The Hartstone benchmarks and some programs recently contributed by PIWG Sweden start to address the control dimension. These benchmarks will be part of the next release of the PIWG suite. Should the linker and/or interface to the run-time system provide detailed information about the location and sizes of the heap, stacks, pure and impure sections, etc., the space dimension could be more easily addressed as well.

It is clear that a large scale collaborative effort between the vendors, PIWG and ARTEWG is needed to truly move beyond the time dimension of run-time benchmarks. But even with some explicit help from the implementation, benchmarking will remain a dirty job requiring the use of (very) global variables and questionable style, sometimes bordering on erroneous programming. Optimizers are becoming more clever than portable software can handle. Ada with its concepts of dependencies, subtypes, and tasking has made new classes of optimization practical. Already, in 1989, the best compilers were starting to break some feature tests of the best suites. This trend can be expected to continue.

What is to be done?

First, feature tests should be de-emphasized. There is no point in attempting ACVC-like thoroughness in this kind of tests. Exhausting testing is clearly impossible and the optimizers are winning that war.

Second, application benchmarks should be emphasized. Good code "Adaptimized" for a class of real applications does not need to trick the optimizer. It does real work, on real data, and produces real results. However, the size and number of these individual benchmarks should be kept low. A dozen or so application benchmarks with an average size of 2KLOC each is a good order of magnitude.

Third, since it is not possible to cover all application domains, benchmark generators should be developed and made publicly available. John Knight et. al. among others have done some promising work in that area [33].

Fourth, it is important to foster research to better understand the time, space, and control dimensions of the compile-time benchmarks described in Chap. 2.

Fifth, even though the fully automated benchmarking of parallel, distributed, and fault tolerant systems will have to await the resolution of complex language issues, the development of nearly portable application benchmarks like the one described by Goforth et. al. in Chap. 5 should be encouraged.

Sixth, the needs of embedded applications should be better addressed by PIWG. PIWGIO package bodies should be provided to handle the most popular microprocessor development systems. PIWG already provides this kind of service with its multiple CPU clock routines.

Last but not least, raw numbers are close to useless without an analysis methodology supported by appropriate tools, such as statistics and graphics packages. Much remains to be done to better present and help analyze the large mass of data that benchmarking suites produce. PIWG's spreadsheet metrics and graphs and the ACEC median tool are notable progress toward more friendly test software.

Much further in the future, one can dream of benchmarking suites of components and of various Ada bindings. However, such benchmarks will only become practical when a high degree of standardization has been reached for this class of software.

To make benchmark execution and interpretation easier, PIWG has proposed that each vendor

provide a "PIWG installation and user's guide" specific for their implementation. Vendors are encouraged to criticize the PIWG tests and help interpret their results in light of their own implementation decisions.

Finally, the opportunity exists for closer collaboration between the major benchmarking efforts worldwide. Already there seems to be some sharing of information between the U.S. ACEC and the British Ada Evaluation System (AES) projects. This is an encouraging development. Its usefulness to the user's community, however, hinges on the overcoming of bureaucratic barriers of the kind that have already so severely hampered the distribution of the ACEC in the U.S. PIWG has recommended that benchmarking software be made freely available [37] and will continue to work toward that end. PIWG will continue to foster the free circulation of ideas, programs and results the world over. Already original benchmarks and suggestions for improvement to the PIWG suite have been received from a dozen or so PIWGs in Europe and Australia.

However, the Performance Issues Working Group is dedicated to performance issues as a whole. As such, we welcome the discussion of results and issues with all benchmark suites. In particular, we have proposed the creation of an ACEC user's group within PIWG. In fact, the ACEC and the PIWG suites complement each other very well. The ACEC is comprehensive, professionally supported and controlled, but of limited distribution. The PIWG suite is simple, "grass roots", totally open, and available to anyone free of charge.

After years of gathering results, PIWG has a significant and unmatched data base of results that is made available for the first time in this special issue. This data will be used for trends analysis and other studies for years to come.

Much has been accomplished since Russell Clapp et. al. published their landmark paper in the CACM in 1986. Today, Ada benchmarking suites allow users to make their run-time performance requirements known to vendors. Vendors use the suites in house to check the improvement they make to their optimizers and run-time systems. The large availability of benchmarking suites probably account for some of the impressive run-time speed improvements witnessed since 1986 for most Ada implementations.

8.3 Other Work

The remainder of this chapter includes two papers describing related efforts in Ada performance evaluation. The first describes the ACEC benchmarking suite, while the second describes the Hartstone synthetic real-time systems benchmark developed at SEI. Following those papers, a discussion of PIWG's future role in Ada performance evaluation is discussed.