Technical Report
CMU/SEI-87-TR-022
ESD-TR-87-173

Factors Causing Unexpected Variations in Ada Benchmarks

Neal Altman

October 1987

Technical Report

CMU/SEI-87-TR-22
ESD-TR-87-173
October 1987

Factors Causing Unexpected Variations in Ada Benchmarks

Neal Altman

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994.

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304- 6145. Phone: (703) 274-7633.

Factors Causing Unexpected Variations in Ada Benchmarks

Abstract

Benchmarks are often used to describe the performance of computer systems. This report considers factors that may cause Ada benchmarks to produce inaccurate results. Included are examples from the ongoing benchmarking efforts of the Ada Embedded Systems Testbed (AEST) Project using bare target computers with several Ada compilers.

1. Introduction

One of the goals of the Ada Embedded Systems Testbed (AEST) Project is to assess the readiness of the Ada programming language and Ada tools for developing embedded systems. The benchmarking and instrumentation subgroup within the AEST Project is running various suites of Ada benchmarks to obtain data on the real-time performance of Ada on a number of different target systems. The purpose of this report is to categorize the factors which cause anomalous results to be produced by the benchmarks. Some of these factors have been observed, while others are more speculative in nature. All these factors should be understood if accurate, comparable, and repeatable performance data are to be produced by Ada benchmarks.

1.1. Objectives and Methodology

The benchmarking and instrumentation subgroup has focused its efforts on measures of run-time performance rather than on the performance of compilers. Our intention is to provide two useful types of information to the mission-critical computer resource (MCCR) community. First, the results of benchmark tests will provide information on the run-time performance of specific instances of Ada compilers, computer hardware, and support software. Second, the methodology used to generate the specific results will be disseminated so that other configurations can be tested in the same way.

Our benchmarking strategy consists of two complementary efforts. First, the work on real-time benchmarks from outside sources is monitored, and, where appropriate, the benchmarks are obtained for use by the AEST Project. The first milestone report of the benchmarking and instrumentation subgroup was "A Survey of Real-Time Performance Benchmarks for the Ada Programming Language" [5]. Second, we will develop our own methods for benchmarking, which include use of specialized timing hardware and methods for confirming benchmark accuracy, as well as more traditional benchmarking programs. Additional information useful for assessing real-time Ada systems will be generated by the Ada real-time experiments subgroup of the AEST Project.

1.2. Facilities

The benchmarking and instrumentation subgroup uses the AEST Project's Real-Time Laboratory to perform its work. Hardware and software components for this facility are still being acquired, with an emphasis on computer systems which are likely to be used by MCCR contractors. At present, the laboratory uses two host/target configurations. The host computer is equipped with a time-sharing operating system and development software. The host is used to enter, compile, and (sometimes) test Ada programs in a feature-rich, user-friendly environment. The target computers are bare machines that receive programs from the host via down-line loading and execute those programs in a dedicated manner. The target computers, having less overhead in systems software, are a more predictable environment for a real-time Ada program.

In its current configuration, the Real-Time Laboratory uses Digital Equipment Corporation (DEC) MicroVAX computers running DEC's VMS operating system for its host systems. Target systems include DEC MicroVAX computers and Motorola Computer's 68020 microprocessors. The VAX computer family is frequently used as a host computer, and its use in the Real-Time Laboratory allows the use of a common set of software tools with a number of Ada compilers. MCCR use of the VAX architecture for target systems is less common, but was undertaken as an interim step because of the availability of the hardware and software and the availability of in-house expertise with the hardware. The MC 68020 is more frequently used in MCCR applications.

Two additional target systems are under consideration: the MIL-STD-1750A microprocessor and several U.S. Navy standard computers.

Currently, the Real-Time Laboratory uses DEC's VAXELN Ada for the MicroVAX target and SD-Ada and VADS Ada for the MC 68020. The TeleSoft TeleGen2 Ada cross compiler is being procured for use with the MC 68020.

Personal workstations of AEST project members are linked to the Real-Time Laboratory via a local area network (LAN) and appropriate software. In addition, dial-up connections allow use of both host and target computers from off site.

1.3. Progress to Date

The initial work of the benchmarking and instrumentation subgroup was a survey of the available work in Ada benchmarking [5]. Based on this survey work, we felt that some of the available benchmarks were sufficiently relevant and should be acquired. The AEST Project obtained benchmarks from the University of Michigan and the Performance Issues Working Group (PIWG) of the Association for Computing Machinery's Special Interest Group on Ada (ACM-SIGAda). The University of Michigan benchmarks were selected for initial runs on the available hardware and software.

The University of Michigan benchmarks were run using the equipment of the Real-Time Laboratory as it became available. At first, Ada compilers running under time-sharing operating systems were tested: DEC VAX-Ada VMS running under VMS on the MicroVAX, DEC VAX-Ada

running under ULTRIX on the MicroVAX, and Verdix Ada running under ULTRIX on the MicroVAX. The next tests used the first cross compiler and target available, DEC VAXELN Ada running on a bare MicroVAX. Work has just begun with the Systems Designers Ada compiler and with a cross compiler version of Verdix Ada, both for the MC 68020 target. The MC 68020 is still being installed and tested.

The original plan for this report was to simply enumerate the results of the benchmarks performed to date. However, during runs of the University of Michigan benchmarks, unexplained variations occurred in running benchmarks — variations which gave rise to *negative* times for certain benchmarks. Review of previous run tests revealed that negative time values could be found in other runs of the benchmarks, including those derived independently by the University of Michigan.

Study of the causes of negative times showed that the problem was related to the structure of the benchmarks rather than to a simple error in a benchmark. To time small Ada constructs with the Ada CLOCK function, benchmarks had to be written which looped, performing the tests many times. The looping control statements add time to the benchmark, which is factored out by subtracting the time required by a second control loop that contains the loop control statements but not the Ada construct being tested. The assumption is that the times obtained from the two loops will vary only in the amount of time required by the benchmark test item (the object of interest). This assumption was unfounded in many instances of real systems.

Not only were the negative time benchmarks wrong, but the problems causing their result were also affecting the more reasonable results. As a consequence, increasing amounts of time have been devoted to examining the assumptions underlying the benchmarks and checking the accuracy of benchmark results. This report summarizes our work on variability to date. It includes topics that we have not fully analyzed. Further investigation will continue, and we plan to provide additional information from a variety of Ada compilers and targets.

2. Variability Described

Variability is used in this report to describe distortions in the benchmarks which prevent their being truly representative of the systems they purport to describe. In instrumentation, the term *measurement error* is used. Variability here has some of the flavor of measurement error, and also includes effects of bad conceptualization and poor implementation. Variability is the difference between what is intended by a benchmark and what is actually produced.

The usual purpose of benchmarking is to provide a basis for comparison between several *functionally equivalent systems*. Systems can be hardware, software, or a combination of both. Functionally equivalent is used here to mean that two or more systems are considered to be comparable enough that they can perform the same function. Of course, benchmarks are often useful in determining that systems are *not* functionally equivalent. For example, benchmarking can demonstrate that System A cannot perform a task rendezvous as quickly as an application requires, while System B can. The two systems are not functionally equivalent for implementing the application.

Certainly benchmarks, like statistics, can be distorted to form unfair or invalid comparisons. Sometimes this is done deliberately, to prove a point or to score a commercial advantage. We are not interested in benchmarks of this sort. Benchmarks can also be misleading because they inadvertently fail to paint a true picture of the system being studied. It is this kind of inaccuracy that we describe in this report.

Benchmarks can be distorted in many ways. A partial list of major causes of variability includes:

- Measurement of the wrong quantity
- Design or implementation errors
- Interference from side effects

This report is primarily interested in interference effects, but each of the other major causes is briefly discussed below.

2.1. Measurement of the Wrong Quantity

Benchmarking is a purposeful activity which aims to quantify some interesting characteristic of a computer system. A benchmark is a measurement tool in the same sense as a ruler or a thermometer. However, many of the interesting characteristics of an information processing system are not directly measurable because they are not precisely defined. We may wish to measure processor power or accuracy, but these useful and general characteristics are, upon close examination, made up of many small, individual units.

Since there is no universal definition for measuring the larger quantities, benchmarks must act on a much lower level. Decisions must be made about which small, presumably discrete, characteristics will be measured. These "objects of interest" are what the benchmark actually measures. In many cases, a synthesis of the smaller measurements into larger and more interesting quantities is also attempted.

Measurements of the wrong quantity can occur during the selection of objects of interest, often due to a misunderstanding of their relationship to the abstract quantity which is wanted. For example, in a system that generates graphic images, the rate at which graphical views can be generated is, in part, the function of how quickly the computer can perform graphical transformations. Transformations depend on the rate at which arithmetic operations — typically multiplications of floating point numbers — can be performed. A benchmark that measures the rate of multiplications could be devised to measure graphic generation rate. However, if the test fails to consider the required precision of results, benchmark results may predict the image generation incorrectly by using the wrong internal representation of the floating point numbers. The benchmark will have been executed correctly but, through design failure, cannot be used to predict accurately.

The process of deciding what quantities to measure can be considered in detail. Here, however, it is sufficient to mention that a dichotomy exists between the synthetic load benchmark and a focused test:

- A **synthetic load** benchmark measures the throughput of a system doing some typical task (e.g., Whetstone, Dhrystone). The time taken for individual operations is not broken out from the total benchmark. A synthetic load is typical or representative of some interesting class of operations.
- **Focused tests** are used to study a specific object of interest. For Ada, the time taken to perform a rendezvous is of interest, and a test can be devised to measure the rendezvous under a set of typical conditions.

Synthetic load benchmarks offer the advantage of providing a built-in synthesis of a series of small tests into a single measurement, but their results are only typical of a specific range of applications. Focused test results can be more widely applied, but synthesis of individual tests into easily comprehended and representative scores is difficult. Focused test results are useful for bottleneck analysis in which a score for a given test must exceed a stated threshold if the system is to be considered suitable for an application.

Both kinds of benchmarks may measure the wrong quantity. Synthetic load benchmarks provide some resistance to measurement error of any individual quantity by their size and averaging effect. However, because synthetic load benchmarks are complex, analyzing them for variability is difficult. Focused tests may be stringently checked on a test-by-test basis, but variability that occurs can have a very large and misleading effect on individual tests.

2.2. Design or Implementation Errors

Design or implementation errors occur when the benchmark fails to measure the desired quantity because of a failure of planning or incorrect execution of the plan. After the object of interest for a benchmark has been identified, the benchmark must be constructed. In most cases, a benchmark is a program. Often, the benchmark is part of a series of similar benchmarks and can be prepared as part of an existing test harness. The benchmark represents a translation of a need for a conceptual quantity to an instrument that will measure the object of interest. If the translation is incorrect, the resultant measurement will not reflect the quantity it is expected to.

As an example, assume that an Ada user has identified the need for a benchmark to measure the execution speed of the multiplication of a series of floating point numbers contained in an array. It is known that some of the target systems will be designed for vector operations, and that the Ada compilers will produce code to facilitate the use of this special hardware (such hardware is to be found on supercomputers such as those offered by Cray Research and selected smaller systems). In a vector machine, operations on matrices may be performed in parallel — different parts of the array being processed at the same time, providing increased execution speed. To simplify the benchmark code, successive array elements will be multiplied. The program uses the construct:

```
for INDEX in (A'FIRST + 1)..A'LAST loop
  A(INDEX) := A(INDEX) * A(INDEX - 1);
end loop;
```

This series cannot be vectorized because the execution of each statement depends upon the results of the previous iteration. The benchmark will not measure the effects of the high speed vector hardware and will, therefore, be in error. A corrected version:

```
for INDEX in A'FIRST..(A'LAST - 1) loop
  A(INDEX) := A(INDEX) * A(INDEX + 1);
end loop;
```

can be vectorized and will use the vector feature of tested machines.

Environment-specific influences (e.g., the effects of a particular optimizing compiler) are not considered here, although a well-designed benchmark will anticipate and overcome such influences.

2.3. Interference from Side Effects

Interference due to side effects are distortions in benchmark results caused by factors external to the benchmark itself. For example, a software benchmark is normally translated by compilers, linkers, and loaders, all of which affect the performance of the benchmark in ways that are not easily accounted for.

The following list summarizes the interference effects discussed in this report:

- Timing Anomalies
    - insufficient clock precision
    - variations in the clock
    - clock overhead
- System Software Effects
    - effects of program translation
        - translation anomalies
            - optimization
            - asymmetrical translation
            - multiple translation modes
        - placement of code into memory

- effects of the run time
    - elaboration
    - memory allocation
    - garbage collection
- operating system effects
    - overhead
    - periodic and asynchronous events
    - co-processes, scheduling, and priorities

- Hardware Effects
    - processor design
        - pipelining
        - multi-processor and distributed architectures
    - memory speeds
        - speed of main memory
        - cache memory
            - cache design
            - triggering the cache
        - segmented memory designs
    - inter-machine variability
        - individual variation
        - minor differences in hardware/firmware

2.4. Format of the Discussion of Interference Effects

Chapters 3 through 5 examine individual causes of interference. Each entry in the summary list above is examined in order. First, the effect is described. Where available and appropriate, examples are provided. Examples are normally drawn from our direct experience, but other sources are sometimes used; interference effects discussed in this report are not limited to those actually encountered. The sections entitled **Detection** and **Solutions** discuss how variation may be identified and avoided.

Since most of our work with Ada benchmarks to date has been with the University of Michigan Ada benchmarks, Chapter 6 discusses the methods used by Clapp et al. for controlling variation.

Terms used frequently in this report are defined below.

CLOCK — The Ada CLOCK function provided in package CALENDAR. Since CLOCK is included as part of the Ada standard, many benchmarks use it for timing.

Object of interest — The specific element of the real-time system under test. Normally, the object of interest is the theoretical quantity which should be measured (e.g., speed of dynamic storage allocation). In many cases, the object of interest is chosen to correspond exactly with some Ada construct, as in "measure the time required to allocate a record with the *new* operator."

Dual loop benchmark — A commonly used technique for measuring the execution speed of small objects of interest with the standard Ada CLOCK. A single test will consume too little time to be accurately measured, so the test is repeated many times, and the time is taken before and after the execution of the series of tests. Normally, the multiple tests are performed using a loop construct. The overhead of the *test loop* is removed by subtracting the time obtained for *control loop*, which contains the loop control statements but excludes the test.

Test loop — Used in dual loop benchmarks to perform the actual test.

Control loop — Used in dual loop benchmarks to factor out the overhead time of the loop control statements.

Precision — Used to denote the accuracy with which an observation can be made. Normally, precision describes the exactness of the CLOCK.

3. Timing Anomalies

Timing anomalies cause interference in benchmarks through the mechanism used for timing the benchmarks. Many benchmarks measure the time required for a selected operation. These times are normally collected by using the standard clock provided by most modern computers. Such a clock is an intrinsic part of Ada, and it is reasonable to assume its presence in any validated Ada compiler. However, the precision of the clock may not be high. In such cases, benchmarks that depend on the Ada clock routines may produce imprecise results.

Examples

Examples are provided below under individual topics.

Detection

The University of Michigan benchmarks check the CLOCK function provided by Ada in several ways. The benchmark suite includes explicit checks to verify overhead for important time routines, and tests for periodic intervention by other processes by means of a second differencing technique.

We plan to test for timing anomalies using specialized hardware (e.g., a logic analyzer).

Solutions

The University of Michigan benchmarks use the Ada CLOCK function provided with the Ada system. Their use of a dual looping benchmark design is intended to avoid the problem of insufficient clock precision, which is explored further below.

We decided not to depend totally on the standard CLOCK function and have been examining the use of more precise timing tools. To date, we have acquired and begun preliminary work with a vendor-supplied real-time clock for the MicroVAX target which is more precise and less prone to interference. Use of this clock requires writing appropriate calls as part of the benchmark. This technique is not general purpose, however, because one real-time clock is required for each target system, with a unique program interface to each clock.

We also have a logic analyzer for use in timing. The unit can be attached to the hardware bus or, in some cases, directly to the CPU. The logic analyzer provides a timing resource that is independent of the system under test and that is very fast compared to normal Ada clocks. At the cost of some effort in identifying measurement points, logic analyzers (traditionally used for hardware development and testing) can be combined with software tests to make timing measurements.

3.1. Insufficient Clock Precision

Insufficient clock precision occurs when the resolution of the benchmark's timer is not sufficient to measure the time taken to execute the object of interest with the desired degree of exactness. This situation often occurs when the standard Ada CLOCK function is used to time the benchmark. Consider the precision of an average wristwatch that displays time to the second. It is sufficiently precise to measure and coordinate certain human activities (e.g., commuting, meetings, meals, and social events); it is not precise enough to resolve a close finish at a swimming meet.

Two factors determine the suitability of a clock. First, the length of time being measured is important. The duration of benchmarks varies widely, from fractions of a second to many minutes. The clock must be able to measure the full duration of the benchmark. Second, the desired accuracy must be considered. The accuracy is conceptually simple to measure, although many practical obstacles often arise.

Assume that the clock measures time in single increments of t. A benchmark which requires a constant time to execute will (assuming no other complications) always return a time of $n \times t$.[1] Expressed as a percentage, the variation of the observation is $\frac{100}{n}$, with the uncertainty being one clock increment (or tick). This percentage is the uncertainty of the observation. Since the precision t cannot be changed for the CLOCK function, the accuracy is responsive to n. Selecting an acceptable percentage is an arbitrary choice, but figures of about 5% or less have achieved a certain respectability as a threshold value. A lower tolerance is even more arbitrary. It is sufficient to assume that below a certain level, other issues, such as the time required to run the benchmark, will be the deciding factors in running the benchmark. For a 5% uncertainty, t must be one-twentieth the duration of the shortest benchmark (or, put another way, n must always be at least 20).

For focused tests, the precision of the Ada CLOCK function is larger than the time required for many of the objects of interest. The benchmark must either average many repetitions of the object of interest or migrate to a more precise timer.

Examples

Virtually every University of Michigan benchmark is written with the assumption that the Ada clock's resolution will be large (or coarse) relative to the time required for the execution of the object of interest. Each of the Ada compilers we examined has used a clock rate which is not precise enough to measure the discrete events considered interesting.

[1] For simplification, this discussion assumes that the benchmark always runs in synchronization with the clock. In many real-world cases, no synchronization occurs, and the time would then be either $n \times t$ or $(n + 1) \times t$. This fact does not materially affect the discussion.

Detection

The precision of the Ada CLOCK function can be determined by inspecting the documentation for the Ada system being used. The *Reference Manual for the Ada Programming Language (LRM)* [1, p. 11-13] defines SYSTEM.TICK as "the basic clock period, in seconds." SYSTEM.TICK is the resolution of the clock, the smallest unit of time which can be measured by the CLOCK function.[2] The same details can be obtained from a simply constructed Ada program.

The other timing devices considered in this report — logic analyzers and real-time clocks — provide similar specifications of accuracy through their documentation.

Solutions

The University of Michigan benchmarks control insufficient clock precision by using a dual loop design, i.e., by repeating the test many times and taking the time only before and after the run of tests. This solution introduces overhead from the loop control statements, which is factored out by running a test loop and a nearly identical control loop that does not contain the object under study.

Dual loop design also has some interesting side effects that illuminate certain important problems in benchmarking which might not otherwise be evident. The logical design of a two loop arrangement suggests that if *Loop T* is a superset of *Loop C*, then the time difference t_d defined by $t_d = t_T - t_C$ should always be zero or positive. For many tests that were run using the DEC VAX Ada compiler, this was not the case. Thinking about the reason why led to this report.

It is important to note that the benchmarks which produced the negative times are as valid as the benchmarks which produced positive times. Both groups were affected by the bias, which was sufficient in some cases to overwhelm the extra time consumed by the test loop. Indeed, the negative values served the valuable function of alerting analysts that further investigation is needed.

The actual causes of the negative times are discussed in Section 4.1.1.2 below.

Software Vernier

A refinement to the dual loop technique, *the software vernier*, is designed to increase measurement precision when used with the standard clock. The general format of a dual loop benchmark is used, but initiation of a timed loop is delayed until the standard clock increments. The loop executes a set number of repetitions, then a tight code loop waits for the clock to change, while incrementing a counter. The counter is used to adjust the row loop time, eliminating the time not actually used by the benchmark, which would normally be counted as part of the benchmark time.

---

[2] Note that DURATION'SMALL, another quantity of time, is the smallest *representation* of time, and "need not correspond to the basic clock cycle, the named number SYSTEM.TICK" [1, p. 9-10]. DURATION'SMALL normally is a great deal smaller than SYSTEM.TICK and should not be confused with it. This is true of each of the Ada systems we have examined to date. As an example:

> In VAX Ada, DURATION'SMALL is 2^{-14} seconds, or approximately 61 microseconds. The value does not correspond to the value of the named number SYSTEM.TICK, which is 1.0^{-2} seconds (10 milliseconds) in VAX Ada. (SYSTEM.TICK represents the smallest unit of time used by VAX/VMS in its time-related system services.) [4, p. 9-14]

The software vernier uses a tight loop as an auxiliary timer to increase clock precision. It assumes that the number of loop repetitions can be converted to times using an empirically derived time value from a calibration test. This assumption is only true when textually identical loops do in fact require the same amount of time to execute, an assumption which does not always prove to be true.

We intend to use dedicated timing hardware to solve the problem of insufficient clock precision.

3.2. Variations in the Clock

Variations in the clock are deviations between the value read by the CLOCK function and the time we would obtain from an absolutely correct timer, were one available. Clock variations impact the benchmark results by introducing errors in benchmarks whose results include time values.

Ada defines two data types for time: STANDARD.DURATION and CALENDAR.TIME. TIME is used for storing a calendar type date and time consisting of a year, month, day, and time of day; it is anchored to the normal Julian calendar. A DURATION is a span of time, in seconds, which is not fixed to any particular zero point. The CLOCK function returns a value of type TIME. Subtracting two TIMEs yields a DURATION. Benchmark results are normally spans of time (i.e., DURATIONs).

Most computer architectures include two kinds of hardware for timing. Computer processors use high rate oscillators for instruction execution. We use the term *system oscillator* for this hardware. System oscillators typically pulse several times during the execution of each instruction. Normally, these pulses are not counted, nor can they be detected by an application running on the processor. A second set of timing hardware is used for providing time information to programs using the computer (both system and application programs). This timer is the *system clock* and, in most systems, is the source of the time provided by Ada's CLOCK function.

Details of the implementation of the system clock vary widely between computer architectures. A "typical" system clock consists of a source of timing pulses (internal or external[3]), a method of counting the pulses (normally software or firmware), and an interface to allow the time values to be read by the software.

The varying demands of timing often prompt computer manufacturers to provide a number of different clocks for a given computer, representing a number of price/performance combinations. It is common for two or more of these devices to use the same interface, and hence completely different implementations of hardware appear as the system clock in a manner that is transparent to the user. It is fair to say that most modern computer architectures provide a standard system clock, and the standard system clock is selected to be an economical source of time rather than the most exact timer available. It is also fair to say that the benchmarker must assume that only the standard clock will be present when preparing generic benchmarks. Even in cases when

[3]A typical external source is the AC power input; in the U.S., AC provides an accurate 60hz signal.

more accurate timers are present, it cannot be assumed that they are automatically used by the Ada system under test.

The system clock can vary from the true time in several ways. First, the clock may *drift*, gradually running slow or fast when compared to some accurate reference standard. Second, the system clock may *jitter*, speeding up or slowing down temporarily, but tending to catch up at some later time. A jittering clock tends to be more accurate when measuring longer time spans, as the jittering evens out, while a drifting clock will vary more from the reference clock as time passes. Of course, an actual clock may both jitter and drift.

The sensitivity of a given benchmark to time variation depends, in part, upon its duration. A short duration benchmark will tend to be more sensitive to clock jitter, while both short and long duration benchmarks will suffer from the effects of drift. If the drift rate is constant, then the percentage error will be about the same. The effects of drift become more obvious with a long duration benchmark.

Of course, no clock is perfectly accurate. What is desired is a clock whose precision is sufficient to allow good measurement. Measuring clock variation would appear to require a second timer for comparison.

Examples

We have not tested for clock variations.

Detection

We intend to use dedicated timing hardware to solve this problem, based on the assumption that a dedicated tool (such as a logic analyzer or a real-time clock) is sufficiently accurate so that clock variation will be negligible. To a certain extent, the assumption will be taken as the solution (i.e., testing for variation in the secondary clock is not planned).

Solutions

Where clock variation is large enough to matter, the most reasonable solution is to substitute a more accurate clock. This substitution sacrifices generality if the clock cannot be integrated into the Ada run-time system via a call to CALENDAR.CLOCK (either the library unit provided by the Ada standard or an overloaded definition). In benchmarks that have been tailored to assume a slow clock (such as the dual looping scheme), the form of the benchmark might have to be altered to take advantage of a better clock, even if it can be accessed via a call to CALENDAR.CLOCK.

3.3. Clock Overhead

Clock overhead is the amount of time that taking a time measurement adds to the test. Reading the clock always takes time, which must be considered in any measurement. However, if the time can be taken quickly, relative to the event being measured, the effect of clock overhead is negligible.

Where the clock overhead is constant, its effects may be removed from the raw benchmark time by a simple subtraction. If the overhead for a clock call is variable, the problem becomes more serious.

Examples

The University of Michigan benchmarks measure clock overhead. The published results show that, using the software clock provided with several Ada run times, clock overhead adds time to Ada benchmarking. We reran the benchmarks on several other configurations and, unsurprisingly, reached the same conclusion.

Detection

Clock overhead for the Ada CLOCK function can be measured by calling CLOCK successive pairs without intervening statements. Any difference between the measured times is the overhead of the clock. Where the clock overhead is less than the precision of the clock, the difference will be either zero or a positive value (one can reasonably expect this to be one SYSTEM.TICK).

Solutions

The University of Michigan's use of a dual loop test will intrinsically factor out the clock overhead when the control time is subtracted from the test time (assuming that clock overhead is a constant), without the need for an explicit subtraction.

We found this approach to be only approximately accurate and plan to use some fast, dedicated timers for software timing.

4. System Software Effects

System software effects are uncontrolled or unexpected side effects of using tools to perform benchmarks.

Each of the Ada benchmarks we ran required editing, compiling, linking, transfer and loading of the executable code on the target computers, execution, and monitoring. Each of these steps required intervention from vendor-supplied software. Each of these software components has the potential to affect the running speed of the software.

This observation is not particularly profound — after all, highlighting differences between program development environments is an important reason for preparing and running benchmarks. However, some of the decisions taken by the system software may arbitrarily change benchmark results, causing a benchmark to show optimal or suboptimal results in what appears to be a random fashion. Even worse, system software effects may never be detected, leading to false reliance on arbitrary observations.

As an example, during any run of a computer program, executable code is eventually assigned to locations in the computer's address space. In some computer systems, the placement of code into memory affects how fast code and data can be accessed, changing the time required for a benchmark. Since a decision like memory placement is based on a large number of factors, many of which are deeply buried in system software, not only is the location not predictable, there can be no reasonable expectation of repeatability except where systems are strictly identical — a daunting task when one considers the complexity of configuring even a modest computer system. System software can introduce uncontrolled variability which has the potential (at least) to overwhelm the values being sought for by the benchmark.

Examples

Examples are provided under the individual sections below.

Detection

Detecting system software effects consists of two major tasks: first, the general detection of an undesirable effect, and second, the isolation of the specific cause. For example, a benchmark that measures the time required to assign values to a variable may be affected if the assignment is recognized as invariant and removed from the body of the loop. General detection would be realizing that the test results are unreasonable. Isolation would be recognizing that the Ada compiler changes the code by moving the assignment outside the timing loop.

A primary consideration for general detection is the need for methods which represent economical use of the experimenter's time. Preemptive techniques are sometimes employed — when the benchmark is written, unwanted system software effects are avoided by anticipating their action and avoiding their activation. In a preemptive scheme, an invariant assignment is disguised by placing the assignment into a subroutine which is compiled after the main body, avoiding the optimization of the code. Anticipation is a useful technique, but it assumes that all the system software effects can be anticipated. It should be supplemented by additional detection methods.

Our work with general detection currently centers on two techniques:

- Reasonableness criteria
- Calibration routines

Applying reasonableness criteria is a somewhat formal application of common sense. It is not reasonable, for example, for a benchmark to produce a negative time value. Unreasonable results should trigger further investigation. Calibration routines are tests whose expected results are known prior to execution. We have written calibration routines to test the dual looping scheme assumption that two identical loops should require the same amount of time to execute. The results demonstrate that this assumption is only true in an approximate sense. For example, in a VAXELN system, two loops of a specific format take the same amount of time plus or minus six percent.

Isolation of specific causes has not been generalized. We currently lean on the use of tools provided by the program development environments. The following methods have been used for isolation; comments pertain to the DEC VAX Ada environment Ver. 1.3 and VAXELN Ada Ver. 2.3:

Compiler listing — Compiler listings are quite useful when they include machine language translations. However, cases of further optimization have been noted which limit the effectiveness of this technique (i.e., the compiler listing is not always identical to the executing code).

Load maps — Load maps provide detailed information on the location of data and code, the sizes of modules, the location and source of included system routines, and similar low-level details. Load maps can be extremely useful in determining how translating of Ada code has been performed. We have had poor success with load maps due to a lack of low level detail in the maps examined and the use of virtual memory, which obscures the actual placements of some code and variables in memory.

Load module dump — The fully linked software can be examined statically before execution begins. This can usually be done on the development system. Dumping the load module entails translating the machine code to human readable form. Facilities for this purpose range from simple direct translations from binary to display characters to sophisticated interpretation. The AEST Project has had poor success with this technique because the available facilities for load dumping were rudimentary.

Dynamic machine code — Observation of executing code via a debugger has proven useful. Viewing the Ada source code being executed is helpful, but the facility to view machine instructions is often needed. In general, dynamic debugging is time consuming but powerful. Limitations include: the need to simplify the execution of benchmarks to reasonable sizes; interference with timing caused by the need to slow the benchmark to human speeds; and the difficulty of interpreting low level details (e.g., system calls) of the execution of the program under the Ada run time.

Solutions

Unfortunately, there does not seem to be any general solution to the problem of system software effects. Only the most intimate knowledge of the internals of system software will allow absolute predictability of the output of the chain of system software which produces the executable benchmark code. This must be combined with an expert knowledge of the hardware/software execution environment in which the benchmark will be run.

Benchmarks should be written to control for as many system software effects as possible. However, since useful benchmarks are comparative tools, some limit must be placed on efforts to prepare benchmarks for any particular machine. The addition of calibration routines will assist in isolating idiosyncratic behavior of individual program development environments. During the execution of benchmarks, the tester must be alert to the possibility of unreasonable results and must vigilantly disqualify or explain unbelievable values.

Even if system software no longer has unexpected results, one issue will remain: should benchmarks be tailored to produce the best possible results, typical results, or worst-case results? Perhaps the aesthetically appealing answer is "all of the above," but the scale of effort required is large.

4.1. Effects of Program Translation

Program translation can alter the benchmark by changing the assumed structure of the program. In a simple case, one compiler may optimize away Ada constraint checking when an assignment is known to be invariant, while another compiler will retain constraint checking. If the intent of the benchmark is to simulate a situation where constraint checking is required and expected, the benchmark will be defeated. Optimization can sometimes be deactivated by the programmer, but not in every case.

4.1.1. Translation Anomalies

Translation anomalies occur when the software is translated from language statements to executable machine code. Translation anomalies include optimization and asymmetrical translation.

4.1.1.1. Optimization

Optimization affects the generated code by changing the designed code into a form which was not intended by the benchmark design. A benchmark makes assumptions about the relationship between the source code and the executable code, such as "this loop statement will cause the generation of machine language code that accesses my variable UPPER_LIMIT." The compiler might violate this assumption by realizing that UPPER_LIMIT is invariant and substituting a constant value into the loop, or by placing the value into register storage.

The intent of optimization is to produce more efficient program execution. Optimization is undesirable when it defeats the assumptions or intent of a benchmark. It is important to prevent optimization of the object of interest, since the benchmark will then become an instrument for measuring an unintended quantity. On the other hand, optimizations which do not affect the correctness of the benchmark are often desirable.

Examples

During benchmark testing, modifications to the University of Michigan dynamic object allocation benchmarks were made. Dummy subroutines were added to the heads of packages containing the timed portions of the benchmarks. The intent was to ensure that the dual timing loops were fully equivalent so that the loop's times would cancel out, leaving only the time required for the execution of the object of interest. Examination of listings from the compiler which contained the assembly code showed that the dummy subroutines were being included. Calibration tests showed that the dummy routines were not having the desired effect. Further investigation showed that these dummy routines were optimized away in the subsequent link step, since they were never called during the execution of the program. The optimization was defeated by calling each dummy subroutine to force its inclusion in the running code. The loop timings were made somewhat more equivalent, although run-time variation in timing continued to be observed.

Detection

We suggest the use of reasonableness criteria and calibration routines for general detection. Calibration routines should closely emulate the program structure of the benchmarks in form, size, and duration of execution. When benchmark results are available from several systems, cross comparison of the results can be used to detect markedly different results for further investigation.

Solutions

There is no simple solution to optimization. In preparing general tests, avoidance of undesired optimization can be pursued. Wulf et al. noted:

> The literature contains a fairly large collection of known optimization techniques; these, in turn, fit into a fairly small number of categories. Nearly all of the unpublished, *ad hoc*, techniques known to the authors also fall into one of these categories.... [6, p. 4]

Wulf et al. go on to categorize optimization techniques as follows:

Constant folding	Calculation of constant arithmetic and logical expressions at compile time.
Dead code	Elimination of unreachable code sequences, including [those] due to constant propagation into control constructs.
Peephole	Reduction of short machine code sequences by passing a "window" over the final object code and eliminating or collapsing adjacent instructions where possible.
Expression reordering	Application of valid mathematical laws such as associativity, commutativity, addition of zero, etc., to simplify arithmetic and logical operations.
CSE in statement	Common subexpression elimination, i.e., evaluation of common arithmetic and logical expression (or address calculations) only once within statement.
CSE in basic block	CSE in a single entry, single exit sequence of code.
Global CSE	CSE across control structures such as loops and conditional statements.

Rankordering	Priorities in use of registers for program variables and compiler-generated temporary variables.
Load/store motion	Data flow optimization to and from variable locations.
Crossjumping	Merging of common code sequences at the end of conditional branches.
Code motion	Motion of common code sequences out of conditional branches and motion of code sequences out of loop bodies leaving the behavior of the algorithm invariant.
Strength reduction	Reduction of multiplication to addition, especially in the case of loop indices used in array subscripts.

> Note that there is no universal agreement as to the names of some of the optimizations. We have used common names where possible (e.g., strength reduction) but the reader should not assume that an optimization is either present in or absent from the list on the basis of our name for it. [6, p. 4-5]

Where optimization changes the validity of the benchmark, it can often be avoided by adroit coding. The University of Michigan tests explicitly avoided optimization in several instances:

> Code optimization, however, can distort benchmark results by removing code from test loops, eliminating procedure calls, or performing folding. The benchmark programs must therefore utilize techniques to thwart code optimizers.
>
> The key to avoiding these problems is not to let the compiler see constants or expressions in the loops whose times are being measured. For example, instead of using a **for loop** with a constant iteration limit, a **while loop** is used with the termination condition being the equality of the index variable to an iteration variable. The index variable is incremented by a procedure, the body of which is defined in the body of a separate package. The iteration variables are declared and initialized in the specification of a library package. Since the iteration values are kept in variables (no constants) and the body of the increment procedure is hidden in the body of the package, there is no way the benchmark loops can be removed by optimization as long as the package specification and body are compiled separately, with the body being compiled after the benchmarking unit.
>
> Similarly, the compiler must be prevented from either removing the execution of the feature being tested from the loop, or eliminating the loop entirely from the control loop that does not contain the feature. To ensure that these problems do not arise, control functions are inserted into both loops, and the feature being measured is placed in a subprogram called from a library unit. Again, if the bodies of these subprograms are compiled separately, and after the benchmark itself, the compiler is unable to determine enough information to perform optimization and remove anything from either the control or test loops. These techniques will be evident in the benchmarks described below.

The loops must each be executed *N* times, as discussed in the next section, to produce the desired accuracy. The form of the test loop is

```
T1 := CLOCK;
while I < N loop
  control functions;
  DO_SEPARATE_PROC_F; -- the function F whose
                      -- time is being measured
  INCREMENT(I);
end loop;
T2 := CLOCK;
T^m := T2 - T1;
```

The control functions and subprogram call to increment `I` are included to thwart code optimizers. The control frame would be identical to this except that a separately compiled function DO_SEPARATE_PROC_NULL would replace DO_SEPARATE_PROC_F. [3, p. 762].

We have concentrated our efforts on running existing benchmarks rather than on developing our own. Working initially with the DEC VAX Ada compiler, our experience with optimization has primarily been aimed at detecting its presence and avoiding it where required. To examine optimization, several techniques have been tried. Each has depended on the presence of a suitable tool for the purpose, a need which has only been partially fulfilled in several instances.

- Examine the assembly language produced by the compiler. Assembly language can be obtained easily by using the appropriate parameter during compilation. However, further optimization (pruning of dead code) occurs during the link stage.
- Examine the locations of variables in the executable code. DEC provides optional load maps during program linking (translation to executable code) and building (addition of system software to the linked code for execution on the bare machine). All addresses provided by the map are in terms of the virtual address space. Addresses of variables and routines are only provided if the routines are made externally visible (e.g., by including them in the package specification). In general, the DEC load maps were not detailed enough for us to use effectively.
- Dump the machine code for static analysis. DEC provides at least one utility for this purpose, but the Ada code subjected to this routine has, so far, resisted successful dumping.
- Examine the executing code. DEC's debugging facilities allow examination of source or machine language during program execution. Output can be written to disk for further work. Use of the debugger prevents normal timing and, in practical terms, can only be attempted for small numbers of repetitions of looping tests.

Looking for optimization in programs has proven that optimization can be detected using standard system tools. However, the effort is time consuming. As a result, looking for optimization effects by human inspection is most efficient when it can be performed on a representative sample of a class of benchmarks. Where possible, examination for optimization should be initiated only where explicit tests or reasonableness criteria indicate that it is needed.

We have devoted little formal effort to developing reasonableness criteria or tests. Informal criteria were applied (e.g., it is not reasonable that some times are reported to be negative). Similarly, certain test programs served, in part, as tests of optimization. These tests replicated the looping structure of the University of Michigan benchmarks (a test loop and a control loop),

with the loops made textually identical. Timing values for such a loop should average to zero. Where they did not, further examination was performed, although optimization was not found to be the cause of the problem.

4.1.1.2. Asymmetrical Translation

Asymmetrical translation occurs when equivalent pieces of code are translated differently, depending on some factor other than their semantic analysis. For example, asymmetrical translation occurs if variations are caused by the position of the code in the package as well as the semantic analysis of the program statements.

Examples

The DEC VAX Ada compiler Ver. 1.3 translated the University of Michigan benchmarks so that the first routine in a package contained one less machine code instruction than the following routines in the package. Since the timing and control loops and their dependent routines were packaged together in pairs in several packages (to avoid optimization), with the test routine first followed by the control routine in each package, the test sequence had several fewer machine code instructions in each repetition. For small tests, where the object of interest in the test loop executed quickly or even added *no* extra time to the execution of the test loop, the test loop could take less time to execute than the control loop, giving rise to negative results. Tests which were producing reasonable (i.e., positive) times were still biased by the asymmetrical translation, although the percentage of the bias decreased as the duration of the test increased.

Detection

The tools used to identify and solve the above example are identical to those suggested in Section 4.1.1.1 above: reasonableness criteria, calibration routines, and inspection of the translated Ada program. When the benchmarks failed to produce reasonable results, test programs whose results were predictable were constructed. Then inspection of software was performed in stages, each stage coming closer to the executing code. The asymmetrical translation was detectable from the machine code listings during compilation.

Solutions

Asymmetrical translation as we experienced it is an interesting example of an unanticipated system software effect. Planning for its presence requires foreknowledge of its existence in a specific software environment. In preparing a general benchmark, anticipating the idiosyncrasies of all the systems it will be tested on is impossible. Successful benchmarks are used by individuals other than the creators to test products which were not available when the benchmark was devised. Any solution would tend to be specific to a given system. The most economical approach to solution is to avoid known problems, and to provide strong detection facilities in anticipation of future problems.

It is worth noting that this particular problem cannot be solved by increasing the number of repetitions nor by applying high speed timers, since the bias occurs during each repetition and within the normal context switch of the subroutine calls. (Of course, with high speed timers, selection of the point of measurement is critical. The exact location of the measurement point determines whether the effect causes bias or not.)

The specific solution undertaken for the University of Michigan dynamic object allocation tests was the insertion of a dummy subroutine at the head of each package that was used in the timing loop. Since the subroutine was called before executing the timing loop, it could not be optimized away, and since it lay outside the timing loop, it did not change the benchmark times. The translation of the control and timing loops was now symmetrical, since they were no longer the first routine within a package. Another solution, not implemented, would be to make each routine in the timing loop into a separate package, thereby ensuring each would be the first routine in a package.

General detection and avoidance of asymmetrical translation has not been widely considered. We have not developed a general technique, but including test programs with expected results would seem to provide some promise for detecting asymmetrical translation without an undue investment of effort.

4.1.1.3. Multiple Translation Modes

Some program translators are designed with multiple operational nodes. For small programs, code generation schemes are used that produce efficient code. As the translator exhausts its resources such as table space, less rigorous schemes are used to translate the program.

Translators which use multiple modes offer the advantage of allowing programs of widely varying sizes to be translated by a single translator. However, textually identical segments of code can be translated differently, depending upon their location within a program or simply because of the overall size of the program being translated.

Use of multiple translation modes can result in non-comparable code within a single program, or in a benchmark program whose results, while valid, will not scale up to larger programs. An interesting unintended consequence would be the invalidation of some synthetic benchmark which is constructed of "typical code segments" drawn from production software.

Examples

No examples were observed.

Detection

When the translation mode shifts in a single program, it can be detected by examining the resultant code. For benchmarks, this examination can be concentrated on critical segments. Scale-up effects should be detectable by a set of programs of different sizes which are designed to produce comparable results. Examination of compiler documentation is useful where the documentation is sufficiently complete.

Solutions

Translator modality is a source of error only in those few cases where it occurs in a way that violates a critical assumption made during benchmark design. If, for instance, a mode change occurs between two timing loops, the translation of the two loops will not be identical. In a series of similar benchmarks (e.g., a series which examines rendezvous with increasing numbers of entry points), translator modality can cause a discontinuity in the results. In most cases, trans-

lation modality represents a source of variation that should be captured to fully describe the environment under test.

4.1.2. Placement of Code into Memory

Placement of code into memory, which is handled by the system's linker, loader, or similar utility, can change the speed of execution in ways which are not allowed for in the program design. For example, the placement of the critical segment of code so that it is in faster or slower memory, or on a boundary so that a cache page fault will always occur, will bias the speed of execution. Since the size of system programs and the implementation of memory in a specific machine can affect the locations, the possibility for variation between runs is high.

Examples

A simple calibration routine containing five identical Ada timing loops was run on two target systems. Consistent variation was reported between the different loops, although the individual loops displayed consistent times within the limits of measurement accuracy.

In the case of the DEC MicroVAX computer under VAXELN, one loop of the five was consistently slower. The percentage variation between the slowest and fastest times varied with the number of repetitions and the code size. In one case, using 100,000 repetitions, the largest time was 12% greater than the smallest. The explanation, based on debugger runs and consultation with the vendor, is that the slow loop appears to span a memory page in the virtual address space. The virtual memory hardware imposes the additional overhead which can be seen in the increased time.

The MC 68020 shows a different pattern, with two fast loops and three slower loops. In the 100,000 repetitions case, the slower loops took about 11% more time. We have tentatively identified the cause as the alignment of the loop's executable code in memory. Loops aligned on full word boundaries required fewer memory cycles for execution.

Detection

Detecting the effects of code placement requires: 1) an understanding of the optimal and sub-optimal locations in a given system's physical memory, and 2) the ability to identify where the critical portions of a benchmark are actually being placed. A simple testing procedure is to run a benchmark several times, varying the location of the code in memory for each run. The relocation should avoid normal memory increments (powers of two) to encourage testing of optimal and worst-case locations in memory.

The example cited above was detected by using textually identical timing loops. The execution order of the individual loops was varied into several different patterns. Since the times for a number of repetitions of individual loops were consistent while the times between different loops varied, the difference in times appeared to be a function of the difference between the loops rather than a function of environmental factors. However, this method was not sufficient to identify the specific cause of the variation. The assistance of the vendor and use of appropriate hardware documentation were required for this purpose.

Solutions

This problem has two dimensions for benchmarking. First, it violates the assumptions of the dual loop benchmark method. Fixing this problem for a dual looping test requires anchoring the test and control loops in the same relative positions in memory. Second, it raises the question of capturing the range of variability for the objects of interest for the full range of memory alignments. The question is how to gain representative examples of these constructs in the best and worst cases.

We have not identified a general solution to this problem. Specific knowledge of the hardware and software being used is required.

4.2. Effects of the Run Time

Effects of the run time are caused by the support software peculiar to the Ada programming language. The run-time software may run on the bare machine, or it may be implemented to run under an operating system or executive. If combined with an operating system or executive, the effects of the run-time system are often difficult to differentiate from those of the operating system (executive).

4.2.1. Elaboration

The Ada Language Standard offers the following definitions:

> **Elaboration.** The elaboration of a *declaration* is the process by which the declaration achieves its effect (such as creating an *object*; this process occurs during program execution. [1, p. D-2]
>
> **Declaration.** A declaration associates an identifier (or some other notation) with an entity.... [1, p. D-2]
>
> **Object.** An object contains a value. A program creates an object either by *elaborating* an *object declaration* or by *evaluating* an *allocator*.... [1, p. D-3]

Entities which are elaborated include packages, tasks, types, and subprograms. Elaboration occurs once, when the entity is first executed. Elaboration of dependent units, such as a library package called by the package undergoing elaboration, will occur when the first package to use the library package is elaborated (but elaboration will occur only once for that library package). Elaboration can potentially increase the execution time of the first use of a package. The compiler may optimize the elaboration of declarations [1, p. 10-12], which implies that the exact sequence will vary from system to system.

Examples

We have not explored elaboration formally. Some observation with the debugger of a VAXELN Ada program (VAXELN Ver. 2.3; VAXELN Ada Ver. 1.2) showed elaboration occurring for each package.

Detection

Elaboration forms part of the language standard. Its presence can be assumed in any Ada execution environment. (Optimization may, of course, intervene to remove elaboration on a case-by-case basis.)

Solutions

The University of Michigan benchmarks execute each benchmark timing loop (control and test loop) several times (usually five) and take the minimum value for each set of observations independently of the other. These minimum values are then used to calculate the benchmark time value. The other observations are printed but do not contribute to the benchmark time. This arrangement allows the effects of elaboration to be discarded in cases where they are large enough to change the benchmark times.

Another suggestion is to precall each elaboration unit prior to use to ensure that elaboration is performed outside the timing loop [2, p. 169].

Interestingly, this problem might be exacerbated when a high speed timer is used. If the timer checks a single iteration rather than averaging a series of runs, the effects of elaboration may skew the benchmark. This may be avoided by pre-elaboration or by taking multiple observations *within a single run*. (Of course, multiple runs would simply measure the overhead of elaboration multiple times.)

4.2.2. Memory Allocation

Memory allocation, in the Ada standard, is handled dynamically for many variable types. The exact method of performing the allocation is not specified. It is easy to envision an Ada implementation which obtains memory for allocation in relatively large chunks and parcels them out in smaller units as needed. The speed of dynamic allocation would then vary based on whether the available supply of free memory was exhausted or not.

Examples

No examples were observed.

Detection

Memory allocation may be observed by controlled experiment. Gradually increasing the size of memory requested by an allocation benchmark will detect any break points, where jumps in the benchmark execution times due to the intervention of memory allocation will be observed. Program development tools often allow user control of memory pools which, in turn, provide information on the allocation scheme.

Solutions

Memory allocation may be controlled in several ways. Where possible, the problem may be avoided by ensuring that sufficient free memory is available at the start of the benchmark execution. Where the tests are looped, changing the size of the loops may avoid memory allocation. If the tests are run multiple times (as the University of Michigan tests are), the problem may be

factored out by discarding the largest result numbers, assuming that the repetition count chosen is not a "magic number" that always causes memory allocation to occur within a given portion of the test.

4.2.3. Garbage Collection

Garbage collection is an operation in which the Ada run time reclaims storage space that was previously allocated but is no longer in use. Garbage collection depends on the specific Ada run-time system, which may or may not actually perform the function. Garbage collection is normally a time-consuming procedure, capable of biasing the benchmark results, which may not occur at a predictable time. (Note, for example, that garbage collection may be sensitive to the amount of physical or virtual memory available on a given system, varying when the amount of memory changes.)

Examples

No examples have been observed. DEC VAXELN Ada (Ver. 1.2) does not appear to perform garbage collection.

Detection

A straightforward software test can be devised to detect the presence or absence of garbage collection. Allocation and (implied) release of memory are continued until memory overflow must occur if deallocation does not. The University of Michigan benchmarks include such tests.

Detecting specific garbage collection intervals is more complex because the triggering mechanism can vary widely. Garbage collection may occur upon release of memory, or it may be deferred until the free pool is exhausted. However, appropriate benchmarks can probably be devised.

Solutions

Garbage collection is not a problem which needs a solution as such. If present, its effects must be allowed for in interpreting benchmark results. The benchmarks which it affects must be identified, and, for these cases, garbage collection can be avoided (e.g., by increasing the free memory pool to avoid triggering it), or its presence can be explicitly noted in the results.

4.3. Operating System Effects

An *operating system* is a software entity that exists to provide services for and to control the operation of other computer programs. An important service of most operating systems is providing a means of sharing a computer among several programs simultaneously. An operating system is a facilitator and does not normally do "useful" work on its own. It is one element of data processing that is often omitted from real-time systems, since these systems often need to sacrifice generality and convenience for predictability and speed. It is present in certain categories of real-time systems, often "stripped down" to restrict interference in the application or as a special, limited functionality entity (often called an executive). Where present, as it often has been in early versions of Ada environments, an operating system can cause variation in the execution of Ada programs.

Note that the Ada run time has provided certain operating system functions (such as task scheduling) within the Ada language. It relieves the implementor from the need to provide such services. The presence of the Ada run time does not imply that the effects of an operating system will necessarily be removed, since run times can be implemented using an existing operating system or executive.

4.3.1. Overhead

Overhead is the cost of using the operating system to do business. For example, the periodic update of the system clock is part of the operating system overhead in some systems. For some operating systems, the clock update may be complicated by the servicing of a timer queue which can be used to activate tasks on a given date or at a given time. Peripheral input/output is another example of overhead. If the overhead is constant, it may be considered part of the object of interest. If the overhead varies due to external factors (e.g., how fragmented storage is on a disk volume or how system tuning parameters are set), it is an uncontrolled variable.

4.3.2. Periodic and Asynchronous Events

Periodic and asynchronous events are interruptions in the execution of a benchmark by burst execution of operating system processes. A typical example is the operating system daemons. Daemons perform some needed function on a scheduled or irregular basis, without running continuously. Electronic mail is often transmitted and received by operating system daemons. Normally, daemons operate in bursts of activity followed by periods of inactivity. To the extent possible, periodic and asynchronous events should be deactivated while benchmarking, or their presence should be allowed for. If they provide services needed by the benchmark, however, there is no method for ensuring that a given run will be consistent.

Examples

Entities which can cause periodic and asynchronous events include communications servers, mail daemons, and virtual paging systems.

Detection

Where the effect of periodic and asynchronous events is large, it can be noted in the variability of the raw execution times of repeated benchmark runs. Examination of the operating system documentation and system start-up scripts will provide information on the configuration of event generators such as daemons. Often it is possible to monitor the system execution with utilities to detect the presence of event generators, although certain system processes are sometimes hidden from view and cannot be detected by examination of the running system. The University of Michigan benchmarks use a second differencing technique to detect with a software test the presence of periodic and asynchronous events.

Solutions

The normal method used to control the effects of periodic and asynchronous events during benchmarking is to deactivate the source(s) of the interruption. This seems particularly appropriate for real-time applications, which are normally run in restricted environments to avoid such interference with time-critical functions.

4.3.3. Co-Processes, Scheduling, and Priorities

Co-processes, scheduling, and priorities is a descriptive phrase for the interference which can be caused by Ada tasks and system processes initiated by the benchmarks. Priority levels and the scheduling algorithm determine which processes (including Ada tasks) will be activated, and in what order, during the run of a benchmark.

Normally, it is assumed that only the benchmark will be running during a timing test. However, the Ada run-time system must be minimally active. Other units, such as a downline loader or I/O daemon, may also be required. Their priority and tuning can cause variation in timings.

Examples

During debugging of the University of Michigan benchmarks using DEC VAXELN Ada (Ver. 1.2) under the VAXELN kernel (Ver. 2.3), diagnostic output using Ada I/O that was initiated prior to the start of a benchmark loop significantly increased time values for the benchmark. The I/O to a slow output device (terminal) overlapped and interfered with the program operation.

Detection

Detecting the effects of co-processes, scheduling, and priorities is difficult to generalize. The effects of these factors depend on the particular environment being tested. This tends to make detection the responsibility of the tester rather than the benchmark developer.

Solutions

The problem is normally addressed by running the benchmark as the sole process, at the highest allowed priority, and without activating any co-processes. The example mentioned above was avoided by the University of Michigan benchmarks because they are written to avoid interference from I/O processing by performing I/O only after the completion of testing.

5. Hardware Effects

Hardware effects are caused by the hardware and firmware making up the tested computer system. The variety of potential effects is large, and the following list is only a partial enumeration of the possible sources of variation.

5.1. Processor Design

Processor design encompasses the specific details of how a given implementation of a computer is designed to execute programs and process data. Details will vary across members of a computer family which share a common architecture, and across computer architectures. Occasionally, the same architecture will be achieved by divergent designs (e.g., MIL-STD-1750A processors). Benchmarks may well favor certain processor designs over others by providing instruction streams which are ideal or worst-case for a particular architecture. Thus, the results of benchmarks may not provide figures that represent the performance of actual applications on a given system.

5.1.1. Pipelining

Pipelining is a technique used in processor design to increase throughput by dividing the execution of a stream of instructions into discrete steps which may be performed simultaneously. The details of how pipelining works vary and can affect the speed of the benchmarks. As an example, in some designs a linear stream of instructions benefits from pipelining, while a program jump is handled by interrupting the pipeline.

Benchmarks which are effectively structured for pipelining will execute more quickly than those which are not, even if the programs are functionally equivalent. In contrast, instruction streams can be devised which effectively negate the benefits of pipelining, achieving a worst-case evaluation of an implementation.

5.1.2. Multi-Processor and Distributed Architectures

Multi-processor and distributed architectures — where two or more computer processors are linked into a single real-time system — represent an emerging technology for real-time systems. Some currently available designs are said to be capable of executing unmodified code on multiple processors (e.g., by dividing the program up by tasks). Other systems require explicit assignment of program units to various processors. Synchronization and intercommunication between the processors ensures that the program results match those from sequential execution.

Effective benchmarking for multi-processor configurations must use the characteristics of these systems.

5.2. Memory Speeds

Memory is an element of the hardware which affects the execution speed of any stored computer program. The extent to which a program meets the ideal form for a given machine's memory affects the time required for program execution.

5.2.1. Speed of Main Memory

The *speed of main memory* — how quickly instructions and data may be accessed — is a factor which seems obvious. Unfortunately, not all computers provide memory which runs at a single speed — that is, the speed of the program may be affected just by where it lies in main memory, without reference to any other factor.

5.2.2. Cache Memory

Cache memory is designed to provide high memory throughput at an economical price. Normally, in a caching design, instructions and data are read into a small (expensive) high speed memory store in blocks, as required, from the slower main memory. Once a block of information has been read into cache, access occurs at high speed. Performance with the cache depends partly on how well the placement of program instructions and data fit the structure of the cache.

Cache memory provides some special challenges to benchmarking. Since the capacity of a cache memory is small, the size of a program that fits completely into the cache memory must be small. Many benchmarks, particularly the parts actually timed, are small. Also, since cache design usually allows several distinct parts of a program to be stored and often inserts and removes segments based on usage, tests of context switching, rendezvous, and the like may also be completely contained within the cache. Small benchmark loops are thus likely to fit into the cache while executing, giving a high performance that would not be achieved in programs of a more realistic size.

The effects of cache memory can be divided into design effects, which are variations caused by the system architecture, and execution effects (here called *triggering the cache*), where the benchmark code introduces variation.

Examples

No examples have been encountered so far. The initial test computer, a MicroVAX II, does not employ cache memory.

Detection

The presence of cache memory can normally be determined by examining the architectural summary of a system. Effects can sometimes be detected in tests which expect equivalent execution times between paired segments of code. Where times vary, operation of the cache may be a source of the variation. However, such simple tests are by no means proof that cache effects will be detected.

It is worth noting that when we considered caching as a problem, something in excess of a day was required to resolve the simple question of whether the MicroVAX II being used for

benchmarking was equipped with cache memory. Literature for the MicroVAX I was replete with descriptions of cache memory advantages, while the MicroVAX II literature was ambiguous. Close examination and consultation with knowledgeable sources finally revealed that cache memory for the MicroVAX II was dropped when a high speed memory bus was implemented.

Solutions

Two concerns have been identified with cache memory. First, any benchmark which uses the test loop/control loop design should, ideally, ensure that the use of the cache is identical between the two loops (e.g., if one loop triggers a cache read during each repetition, the other should as well). However, since the loops are of different sizes, this can be difficult to achieve in some special cases. Second, if use of the cache can significantly alter benchmark timings, should the benchmark capture best case, worst case, or some average case?

In general, knowledge of the caching design will facilitate intelligent solutions to the problem. If caching speed is an issue, a simple test might be the testing of a loop (with small sample size) against a program consisting of a large number of identical statements (in essence flattening the loop to a linear code sequence).

5.2.2.1. Cache Design

Cache design — the size of the cache and the number of different cache units — is usually a hardware design feature which is not under the control of the Ada programmer. A cache runs faster because a small amount of data can be stored in high speed memory. Controlling a program for maximum (or minimum) efficiency is not a part of standard Ada, so the loaded program may or may not be optimized for the cache.

5.2.2.2. Triggering the Cache

Triggering the cache occurs when a program references a data location that is not currently resident in a cache memory.

Examples

No examples have been encountered so far.

Detection

See the discussion in Section 5.2.2 above.

Solutions

In a benchmark containing dual test and control loops, the loops should be aligned to cause the same number of cache triggerings. This requires sufficient control over the placement of code into memory, which is provided by the Ada standard. This assumes that the address specified in an Ada program can be used to force alignment to a physical location (normally alignment to a "page" or similar memory unit).

5.2.3. Segmented Memory Designs

Segmented memory designs allow a relatively small virtual address space to use a larger physical memory. Specialized hardware expands each memory reference into a reference to physical memory. Segmentation was used in the design of several popular machine architectures (e.g., MIL-STD-1750A, Intel 80x86). Segmentation forces hidden "memory context switches" and differing instruction sequences for programs that happen to fall across boundaries. Again, the behavior of a given benchmark depends on how the machine code is ultimately structured and is not under the control of the Ada programmer. Optimization can provide much improved timings, while the size and placement of system routines can affect what are otherwise identical benchmarks.

5.3. Inter-Machine Variability

Inter-machine variability is the difference between different machines of the same make and model.

5.3.1. Individual Variation

Individual variation is the intrinsic difference between two different samples of a given system. Obviously, this is a factor when the benchmarks are run on a machine other than the target. It can be argued that software is not subject to this problem but is prone to minor differences (e.g., a patch).

5.3.2. Minor Differences in Hardware

Minor differences in hardware occur when the manufacturer, maintainer, or customer makes hardware changes that differentiate the benchmark machine from the units that will actually be used. A particular problem is that computers often provide many methods of making very subtle hardware changes (e.g., strapping, configuration of components, termination of circuits, engineering changes) that are not documented and whose effects on benchmarkings are not well understood.

Examples

No examples have been noted so far. Tests using "identical" MicroVAXes have not been completed. However, of informal comparisons, benchmarks performed on multiple "identical" targets have not thus far displayed any noticeable differences.

Detection

Minor hardware differences can be detected by simple comparison of results between runs made on different "identical" machines. This has the advantage of ignoring any differences which have no detectable effects. Detection can also be pursued by inspecting components for differences, including alterations after purchase, installation, and repair. Detection constitutes an ongoing process, since the constant upgrade of field serviceable and swapped components is usual for most vendors. However, noting a difference is not equivalent to isolating its effect on a benchmark.

Solutions

There is no solution to this problem. Confidence in the applicability of the benchmarks to all systems may be increased by repeating the tests on multiple instances of "identical" machines. Where feasible, the target being tested and the options configured on it, as well as the versions of software being used, should be specified as part of the benchmark report. Normally, specification to the lowest level (e.g., board revision level or software fix level for all components) is not available.

6. Variation Control Techniques Used by the University of Michigan Benchmarks

The University of Michigan tests [3] focus on the execution speed of specific features of the Ada programming language. This approach isolates relatively small items to allow comparisons of critical interest (e.g., rendezvous versus subroutine call), but has led to some very interesting side effects. In particular, these focused tests characteristically have a very small size, especially the size of the critical segments where the object of interest is tested. A single object of interest often requires very little time to execute — often so little time that the precision of the system clock is larger than the time required for the test.

To test such small objects of interest, most of the University of Michigan benchmarks use a dual loop testing design. A test loop contains the actual test along with control statements to repeat the test many times. The time is taken at the start and end of the loop. A similarly structured control loop is constructed which excludes the test of the object of interest but contains all the loop control statements. Subtracting the control loop time from the test loop time will factor out the overhead of the loops. Dividing by the number of repetitions provides the time required for the object of interest.

University of Michigan Ada benchmarks have solved a number of problems causing variation:

- Timing Anomalies
    - *Insufficient clock precision* as compared to the time taken to execute the object of interest. First, the University of Michigan benchmarks check the resolution of the Ada CLOCK function empirically. Second, based on existing compilers, they assume that the clock will be slow compared to the time required to execute the object of interest. They control this factor by repeating the test many times and by taking the time only before and after the run of tests. This solution introduces overhead from the loop control statements, which is factored out by running a test loop and a nearly identical control loop that contains only the looping and other overhead code, not the actual test. The time required for the control loop is subtracted from the time used by the test loop, and the difference is divided by the number of times the test has been performed.
    - *Clock overhead.* The University of Michigan benchmarks take account of clock overhead in two ways. First, explicit tests provide information on the overhead imposed by the use of standard Ada time functions. Second, the impact of clock overhead on benchmarks is factored out by use of the dual looping test scheme. The benchmarks consult the clock at the start and end of the test and control loops, then factor out the overhead by subtracting the control time from the test time.
- System Software Effects
    - Effects of program translation
        - Translation anomalies
            - *Optimization*. The University of Michigan explicitly protects against optimization by hiding constant variables from view, preventing simplification of loop constructs, and by arranging the order of compilation for similar purposes.

- Operating System Effects

  - *Periodic and asynchronous events*. The University of Michigan benchmarks use a second differencing technique to estimate how frequently operating system processes interfere with the execution of their test program.

    For many University of Michigan benchmarks, a single trial consists of multiple runs of the control and timing loops (five runs in many instances). The minimum value for each control and timing loop is taken independently, and the result is obtained by subtracting these two values. It is assumed that the minimum figure represents the smallest amount of interference from the operating system.

7. Summary and Conclusion

The benchmarking and instrumentation subgroup of the Ada Embedded Systems Testbed Project has examined a number of benchmark suites which are intended to evaluate Ada run-time environments. Work with well-conceived benchmarks has still provided results which were inconsistent or improbable. Increased effort in identifying the causes of variation within the benchmarks has identified potential for variation in clocking, system software, and hardware.

The dream of an ideal, transportable set of Ada benchmarks which can be plugged into any run-time system without modification is an illusion. It may be pursued indefinitely without final result. A more realistic view includes the tester as an active participant in testing and adjusting benchmarks to work with the system. Basic assumptions about benchmarks and their measurement accuracy must be tested before the results are published. Individual benchmarks must be qualified, and variation must be explained.

Some variation in results can be controlled by good design and, where required, by modifications to the benchmark code. The accuracy of benchmarks may be enhanced by:

- Using more accurate timing devices, such as real-time clocks or logic analyzers.
- Constructing calibration routines to identify variation by expected results.
- Executing benchmarks on multiple samples of a given system.
- Developing reasonableness criteria, which trigger further investigation when results are not logically consistent with theoretical values.
- Interactive testing, where the execution of benchmarks is conducted with the expectation that some of the results may have to be adjusted or rejected as non-typical.

The use of dual looping benchmarks, a usual method of overcoming imprecise clocks, has been examined. The method is applicable to all Ada systems which provide the CALENDAR.CLOCK function allowing unmodified benchmark code to be run on multiple systems. The basic assumption that two loops containing the same Ada statements will require the same amount of time to execute has been demonstrated as false for several systems. The accuracy of the method is therefore variable. It remains a strong technique for averaging out the effects of one-time and periodic overhead factors in the execution of the benchmark.

Certain classes of variation can, conceivably, be eliminated by adjusting the benchmark software or system software, or by using external timers. For example, it is possible to envision adjustments to systems to fully validate the assumptions of the dual looping benchmarks. These kinds of variation are, therefore, problems to be solved by adroit application of knowledge.

Other classes of variation are an immutable part of the system under test and present a more philosophical question: what *kind* of result is desired? Should the benchmark strive to present a "typical" result, or is a best case/worst case range more practical? A single value for any benchmark is not sufficient unless it is repeatable.

As a result of this variability, we have redirected our focus from executing suites of benchmarks uncritically on a variety of targets to a more critical role. Future work will emphasize validating the benchmarks as well as executing them. Validation will require software tests, specialized hardware, and logical analysis of the systems on which the benchmark is executed.

References

[1] *Reference Manual for the Ada Programming Language.*
ANSI/MIL-STD-1815A-1983 edition, ANSI, 1430 Broadway, New York, New York 10018, 1983.

[2] Broido, M.D.
Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 30(2):169-171, February, 1987.
This is a technical communication from M. Broido with a reply by Clapp, et al. on Clapp's article of the same title published in *Communications of the ACM*, August 1986.

[3] Clapp, Russell M., et al.
Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 29(8):760-778, August, 1986.

[4] *VAX Ada Language Reference Manual.*
Digital Equipment Corporation, Maynard, Massachusetts, 1985.

[5] Donohoe, P.
A Survey of Real-Time Performance Benchmarks for the Ada Programming Language.
Technical Report CMU/SEI-87-TR-28, Software Engineering Institute, October, 1987.
To be published.

[6] Wulf, W., Feiler, P., Zinkas J., and Brender, R.
A Quantitative Technique For Comparing the Quality of Language Implementations.
July, 1979.
This unpublished report is quoted with the permission of the senior author.

Table of Contents