The ESA Ravenscar Benchmark [⋆]

Romain Berrendonner
Jérôme Guitton

AdaCore
8 rue de Milan, 75009 Paris, France
berrendo@adacore.com
guitton@adacore.com

Abstract. This article presents ERB, the ESA Ravenscar Benchmark. ERB aims at providing a synthetic benchmark comparing the efficiency of various Ada Ravenscar implementations and the RTEMS C implementation featuring the native threading model. ERB is original compared to existing Ada benchmarks, such as the ACES or the PIWG, not only because it is the first Ada Ravenscar benchmark, but also because it provides at the same time measurement of execution times and estimate of the memory footprint of the Ada runtime and stack size requirements. ERB intends to become the standard benchmark for embedded Ada Ravenscar applications. To facilitate this, the European Space Agency and AdaCore plan to release it under the GNU GPL to interested third parties.

1 The ERB Project

1.1 Project Context

The European Space Agency (ESA) has devoted signifi cant resources to the development of a radiation-hardened family of processors based on the SPARC architecture [24] for use in European space applications. This family includes the ERC32 radiation-hardened SPARC V7 processor and the Leon SPARC V8 VHDL model.

The main programming languages currently used for space applications are Ada and C. Various Ada and C compilation tool chains presently exist that target the space market. Some of these tool chains are self-contained, allowing the creation of self-standing embedded applications; others rely on a kernel to provide the complete environment and services needed for the fi nal embedded application. This is particularly true for the C language, which does not provide any facility for concurrent programming; the only way to achieve this is by providing a kernel, such as RTEMS or VxWorks , with a C API.

Several elements come into play when choosing the suitable compiler and/or kernel for a given project: robustness and richness of the tools, quality of the support organization, run-time performance, trustworthiness of the organization behind the tools, etc.

⋆ This work has been funded by ESA/ESTEC contract No. 16962/02/NL/LvH/bj and carried out in cooperation between AdaCore, the Technical University of Madrid and the University of Padua.

In the context of space applications, where reliability of code is paramount, adopting a restricted and standardized model for tasking and concurrency makes concurrent applications easier to understand and implement. The Ravenscar model [7] has now been approved by WG9 and will be incorporated into the new revision of Ada ISO standard specifi es due in late 2005 or early 2006. Such a restricted model is the natural fi eld of experimentation for the Ada-related part of this study and, indirectly also for the C part of it. It aims at providing a high-integrity effi cient Ada implementations standard for real-time systems, at the cost of some restrictions to the standard Ada 95 tasking model. All the restrictions facilitate the schedulability analysis of programs using the model and allow several optimizations in the underlying run-time libraries.

Once a tool chain has been selected, it is important that the software development team be able to easily recognize the run-time performance hot spots of their compiler/kernel so that the appropriate coding guidelines can be adopted for on-board software development.

Last, but not least, in the case of Ada, the engineering team must be able to evaluate the cost of using high-level Ada features (e.g. exceptions, functions returning unconstrained objects, etc.) and gauge the trade-off between their run-time cost and the software engineering benefi ts for their project so as to choose the appropriate software architecture for their space application. In the case of C, the performances you can measure are tightly related to the operating system: the language has no high-level features and the compilation process produces the code in a straightforward manner. Therefore, the performance of a C program is directly connected to the effi ciency of the operating system.

The ESA Ravenscar Benchmark (ERB) [15] [18] [17] project has been awarded by ESA to AdaCore to address those concerns and provide a composite synthetic benchmark targeting Ada Ravenscar applications for the ERC32 processor. ERB also provides a limited set of tests targeted to the C environment on RTEMS for the purpose of evaluating the cost of high-level Ada features against their C counterparts, which have been undertaken by a team of the Technical University of Madrid led by Juan-Antonio de la Puente.

This article aims at presenting the ERB projects, showing how it is different from existing benchmarks, how it is designed, and its intended future.

1.2 Some Prominent Benchmarks

A wide variety of benchmark suites are currently available to developers who wants to quantify the performances of a run-time system; it is not the point of this paper to provide a complete list. However, it is interesting to mention:

– **The SPEC suites [2]:** The Standard Performances Evaluation Corporation develops well-known benchmark suites for C compilers. Existing benchmarking suites target CPU usage on workstations as well as mainstream applications for the Internet, such as web servers, mail server or fi le sharing. Interestingly, SPEC uses a set of high-level applications for measuring CPU usage, rather than doing synthetic micro-benchmark. Applications used by their flagship CPU2004 benchmark

suite comprise for instance implementations of the GZIP and BZIP2 compression algorithms or compilation of GCC.
- **EEMBC [14]:** The Embedded Microprocessor Benchmark Consortium provides suites of benchmarks targeted to embedded systems and C compilers; Interestingly, this benchmark uses both high-level full implementations of common algorithms, as well as low-level, simple, functions.
- **MiBench [20]** is another benchmark suite for embedded systems developed at the University of Michigan; MiBench is very close to EEMBC.
- **lmbench [21]** is a good example of a micro-benchmark suite, used to measure the cost of some specifi c features of an operating system, e.g. process creation.

The previous benchmarks are targeted to C compilers. Two well-known benchmark suites are targeted to Ada tool-chains:

- **PIWG [22]:** The PIWG's goal was to support performance assessment for Ada compilation and execution systems. The original path to achieving this goal led to the creation of a set of benchmark tests for Ada 83 which was developed and distributed during the 80s and early 90s and contains about 80 Ada83 tests.
- **ACES [5]:** In response to the need for Ada developers to better know the tool-chain they were using, both the US DoD and the British MoD funded a joint effort for evaluating Ada compilers. This result of this project has been made later available to the public and is now known as ACES. The ACES uses the dual-loop method to provide reliable measurements of execution times. This allows measuring small chunks of code as well as full algorithms. However, the ACES needs signifi cant work and skills to be turned operational.

Whereas speed and throughput are common fi gures in a benchmark output, memory use measurements are rarely performed. Some benchmark suites give static measures, such as code size; EEMBC is an example, and ERB, as shown below, is too, but part from this simple measure, only one benchmark suite gives dynamic memory behavior among all the benchmarks we have studied.

In software systems, two memory areas are likely to grow dynamically: the heap and the stack:

- **Pennbench [16]** is the only one benchmark suite providing heap usage measurement. PennBench is a benchmark suite for Embedded Java; in that context, the heap usage is a relevant memory measurement. A Ravenscar application typically does not contain many dynamic allocations; In a safety-critical system, heap allocations are typically allowed only in a initialization phase. Moreover, the Ravenscar profi le adds some other constraints on the dynamic allocation; for example, it forbids constructs that require an implicit heap allocation. This is why ERB does not provide this measure.
- **No benchmark**, to the best of our knowledge provides stack usage measurement.

1.3 Why A New Benchmark ?

The European Space Agency realized soon that none of the existing benchmarks were relevant in the context of Ada Space applications. Like SPEC, most of those benchmarks primarily target workstation environments. They are simply not suited for embedded applications. The few benchmarks targeting Ada environments, like the PIWG and the ACES, are not suited either for modern embedded applications: the PIWG do not comply to Ada 95, while the ACES are not Ravenscar compliant.

Unfortunately, the Ada benchmarks are out of date in that respect. Most of the tasking test cases they provide break many of the above restrictions, and would not even compile under the Ravenscar profi le. PIWG test `t000003` is a perfect example of this: it aims at measuring the task entry call and return time when the task is in a package. Of course, it needs to use task entries, which are forbidden by the Ravenscar standard. As a matter of fact, it is not only a matter of removing the test cases that cannot be compiled under the Ravenscar profi le: it is also required to get a new set of test cases precisely targeting the kind of constructs programmers actually use in real-time applications.

In embedded systems, memory is by every respect as important as timing. There is no point in making a careful timing analysis of a system to make sure it is able to answer quickly to critical conditions if the stack of one task overflows on another task space and causes the program to malfunction. One the other hand, fi tting a lot of memory may have a high cost, in terms of cost, weight and power consumption.

To make sure one has enough memory in his application, one needs not only to know the stack size required by each task in the program, but also one needs to know the runtime footprint in memory. This can be defi ned as the code corresponding to functionalities the user code requires. For instance, when using Ada tasking, a program depends on the task switching code provided by the system or the runtime. The order of magnitude of the runtime can be signifi cant: the ORK [13] runtime has for instance a 150KB footprint.

The ERB benchmark therefore provides not only execution time measurements but also an estimate of the stack size required by each task in a test case to execute properly as well as an estimate of the runtime and user code footprints.

Finally, in order to evaluate the cost of high-level language features in terms of memory footprint, stack size and execution times compared to low-level C code, the Agency required that some of the tests would be ported to C for running on the RTEMS system.

This raised several issues. The semantics of Ada is much richer than one of C: protected object, exception handling, tasking, distributed partitions do not exist in C. In order to implement similar features, one has to rely on special-purpose libraries, such as the native or POSIX threading library on RTEMS or the Remote Procedure Calls library. In any event, real C programming would not try to match full Ada semantics, but probably use simpler mechanisms: instead of a protected object, one would for instance rather use System V message passing or queues. All of this makes any comparison between C and Ada particularly delicate. The approach taken in ERB was to write a set of C packages providing exactly the same semantics as the required Ada features. Even if some ad-hoc Ada code could only implement a subset of those features for a particular test case, a real space application would probably benefi t from the availability of such semantics.

Another decision had to be make about using the RTEMS native threading interface against the POSIX threading interface. The native threading model was fi nally selected for two reasons: first, it is the default mode when RTEMS gets installed; second, Ada tasking is part of the runtime, therefore using a separate, probably less optimized, POSIX library would be unfair to RTEMS.

In a nutshell, the ESA Ravenscar Benchmark (ERB) project aims at providing a new benchmark targeting Ada Ravenscar implementations and RTEMS on the ERC32 processor. The choice of the ERC32 processor was obvious for ESA since the Agency has spent significant resources to turn it into a standard for the Space industry. Like the ACES or the PIWG environments, the ERB benchmark uses mostly synthetic tests: each of these tests is designed to exercise a particular, clearly identified, Ada feature. A set of applicative tests are provided anyway to compensate the lack of full-fledged, space-specific application. Example of applicative tests include Cyclic Redundancy Checks and the Data Encryption Standard. Additionally, each test is either designed to provide execution time measurement or memory measurement, making ERB the first multi-purpose benchmark we are aware of. A subset of the Ada tests has been ported to RTEMS using the C language and the native threading model. C code will be written to exactly match the Ada code in the corresponding test cases.

2 Technical Overview

This section presents a quick overview of the technical choices made in the ERB project.

2.1 Portability Considerations

Portability can have many different meanings in the context of a benchmarking project. One has to consider target portability, host portability and environment portability.

Target portability is defined as the capability to use the test suite on a different target processor without significant adaptation efforts. In the case of the ERB contract, the ERC32 was the only mandatory target; AdaCore is nevertheless already considering running it on PowerPC processors. As a consequence, target portability was considered since the early stages of design: in order to facilitate adaptation to other processors, and for cost-efficiency reasons, it was decided to use a simulator. The choice for the ERC32 target was the `sis64` [6] ERC32 simulator. Switching to another processor should therefore be as simple as using the corresponding simulator, adapting the interface between the simulator and the harness program if needed, and modifying the support package used for stack size measurements.

Host portability can be defined as the capability to run the test harness on a different host workstation. The host is used for compiling each test, running the simulator and getting static memory information. The Agency required the test harness to be usable on Linux and Solaris; once again nothing in the design prevents it from being ported to other work environments such as Microsoft Windows. The main harness driver is written in Ada 95, for reliability and portability. Wherever Ada was not a possible choice, Bourne Shell were used instead because it is probably the most widespread and portable interpreter available. It is even part of the Cygwin package on Windows.

The last kind of portability one has to consider is *the environment portability*. In this document, an environment is an Ada 95 Ravenscar tool chain able to compile code for the target processor. The system has been designed so that adding support for a new environment in the harness is not difficult. Support for a particular environment is customized by a set of configuration variables contained in two files. One is global and contains mostly the name of the environment and of the simulator being used; the harness comprises a tool for changing it. The other configuration file is environment-specific and contains all the information required to run the test harness in the considered environment. This includes information such as the comment delimiter in test source code as well as the commands used for compiling each test case and checking the environment. Average users do not need to modify this file. Adapting the harness to a new environment is therefore only a matter of writing those two command scripts, which invoke environment-specific commands. Adapting the support packages to a new environment is no more complicated. By definition, environment-independent support packages do not need any adaptation. Environment-dependent packages all have an environment-independent specification; only the implementation of the package is environment-dependent. In environments which contain a fully-featured Ada runtime, implementation will mostly contain renames of runtime routines. On high-integrity environments that contain a tightly controlled set of runtime features, like GNAT Pro for ERC32, the implementation needs to contain full implementations.

2.2 Timing Measurement

As we said above, most of the existing benchmarks compute metrics on real applications. This approach, inspired by the desire to mitigate the influence of low-level issues and get a global figure, is not very well suited for a fine-grained analysis of the contribution of individual language feature to the overall benchmarking results. That's why the PIWG and the ACES benchmarks take a different approach. The method they use is known as the dual-loop method, and it provides accurate measurements for individual language features.

The Dual-loop Method A complete study of the dual-loop method is beyond the scope of this paper. For a complete description, the reader is welcome to read [5].

The dual loop method is based on a careful analysis of the possible systematic errors related to timing measurements. The two most prominent ones are the jitter, which is basically the random variations of clock precisions, and the quantization error, which is mostly caused by the analog-digital conversion of the physical phenomenon used to generate the clock signal. A program is first executed on the target to estimate the order of magnitude of those errors, which repeatedly counts the number of system ticks in one milliseconds to determine the accuracy of the clock and the jitter.

The dual loop strategy is to repeat each test case a number of times, so that the overall execution time is significantly longer than the possible jitter and quantization error estimates. "Significantly longer" here has been arbitrary set to 100 times longer. In order to do this, the test code is surrounded by two loop: the inner loop just executes the test code a fixed `NCount` number of times. `NCount` is fixed by the outermost loop, and keep increasing[1] until the the execution time of the inner loop meets the condition

[1] The number of counts follows a geometric series: $Ncount_{N+1} = 2 * Ncount_N + 1$ starting with $N_0 = 1$. Additionally, a maximum upper bound is provided to avoid infinite loops.

above. Once it is the case, the inner loop is executed `MCount` number of times with the same `NCount` to demonstrate that the measurement is stable. `MCount` is fixed so that:

1. It is lower than an arbitrary bound to avoid infinite recursion;
2. Each test does not take longer than 30 minutes to execute;
3. A Student's t-test indicates that, to the requested confidence level of 90%, the timing measurements are being drawn from a sample with the same mean.

If the test case ends with condition 1 or 2 above, an error is reported.

The ACES have demonstrated that the dual loop method provides very good results with the compilers they have been using. They have in particular used advanced heuristics to handle compiler optimizations, processor cache issues and memory paging. But one problem remains: if the code being tested has side-effects, the dual loop method reaches some limits. In a sort algorithm, for instance, the first run of the code will execute much faster than any subsequent run that acts upon the already sorted array. Of course it is possible to add in the test case some initialization code to "unsort" the data; but in that case, you do not measure the sort algorithm only. Another issue that appears when using the dual loop method is the impossibility to measure elaboration code, since it is out of the reach of user instrumentation.

In order to address both of these issues, it was decided to implement two additional strategies for timing measurements.

Other strategies The first strategy is very simple: once it is compiled, the test case is just loaded and executed on the simulator, using the simulator timing facility to retrieve the execution time. The harness is responsible for repeating the test a number of times so that it is possible to make statistically sure that the results are meaningful. In our case, this is done by checking that all the measurements fits into a 90% confidence interval. This strategy is called "run-all".

The second additional strategy we provide is known as "run-it-once". It is particularly intended for code using side effect. Instead of surrounding the test code by a dual loop, this method just uses the target environment timing facility to get the execution time of a single execution of the test. The harness is responsible for repeating the test a number of times, so that it is possible to make statistically sure that the results are meaningful using the same kind of proof as the "run-all" strategy.

Common Instrumentation Code Each of the timing test cases is executed with all three strategies. It is possible to have several test cases into a single test source file. The tests have some markers in the their sources that get expanded differently according to the strategy actually being used:

– `<init_loop>`: In the dual-loop strategy, this is expanded into the initialization code for the variables used in the dual loop code, which directly uses the output of `erb_pretest` program. In other strategies, this does not get expanded at all. There should not be more than one instance of this tag in the source code [2].

[2] But having several of them does not harm either, since it would only duplicate some variable initialization code

– `<start_measure>`: In the dual-loop strategy, this is expanded into the first part of the dual loop code. In the run-it-once, this just expanded to some code reading the clock. This is not expanded at all in the run-all strategy. There can be multiple instances of this tag, provided they are separated by an instance of `end_measure`.
– `<end_measure>`: In the dual-loop strategy, this is expanded into the second part of the dual loop code. In the run-it-once strategy, this is expanded to the code reading the clock and printing out the overall execution time. This is not expanded at all in the run-all strategy.

Note that those markers are contained in Ada comments so that unexpanded test cases are still legal Ada programs. C test cases on RTEMS use exactly the same mechanism.

A complete example coming from test `ar_t_a_03` follows. It shows how the markers must be inserted into the test source.

```
-- Test conversions between fixed point type and Float
-- For: Ravenscar
-- Measurement: Timing
-- Based on: ACES ar_cx_conv_fixed_04

-----------------
--
-- Interestingly, we are able to have several tests in the same
-- test case. Here we have:
--
--      (A) Conversion from delta 0.01 to Float 6 digits
--      (B) Conversion from delta 0.01 to Float 12 digits
--      (C) Conversion from delta 0.001 to Float 6 digits
--
-----------------

with Support;       use Support;
with Support_Types; use Support_Types;

procedure Ar_T_A_03 is

   pragma Suppress (Access_Check);
   pragma Suppress (Discriminant_Check);
   pragma Suppress (Index_Check);
   pragma Suppress (Length_Check);
   pragma Suppress (Range_Check);
   pragma Suppress (Division_Check);
   pragma Suppress (Overflow_Check);
   pragma Suppress (Elaboration_Check);
   pragma Suppress (Storage_Check);

   A_Float_1 : Float6 := Simple_Float6_Random;
   A_Float_2 : Float12 := Float12 (Simple_Float6_Random);
   A_Fixed_1 : Afix1 := Simple_Afix1_Random;
   A_Fixed_2 : Afix3 := Afix3 (Simple_Afix1_Random);
```

```
begin
   -- <init_variables>
   -- This tag is expanded to the initialization code
   -- required by the dual-loop strategy

   -- Test (A)
   -- <start_measure>
   A_Float_1 := Float6 (A_Fixed_1);
   -- <end_measure>

   -- Test (B)
   -- <start_measure>
   A_Float_2 := Float12 (A_Fixed_1);
   -- <end_measure>

   -- Test (C)
   -- <start_measure>
   A_Float_1 := Float6 (A_Fixed_2);
   -- <end_measure>

end Ar_T_A_03;
```

Typical output of this test when executed with the dual-loop strategy looks as shown below:

```
--------------------------------------------------------------------------------
| Name       | Min            | Mean           | Sigma       | MCnt | NCnt | Msg |
--------------------------------------------------------------------------------
| ar_t_a_03  | 2.16235590E+0  | 2.16235590E+0  | 0.00000000% |    5 |  127 |  Ok |
| ar_t_a_03  | 2.01235628E+0  | 2.01235628E+0  | 0.00000000% |    5 |  127 |  Ok |
| ar_t_a_03  | 2.16235590E+0  | 2.16235590E+0  | 0.00000000% |    5 |  127 |  Ok |
--------------------------------------------------------------------------------
```

- `Name` is the name of test case. There is exactly one for individual test inside the test case;
- `Min` is the minimal execution time of the test;
- `Mean` is the average execution time of the test;
- `Sigma` is the standard deviation computed on the series of test case;
- `NCnt` is the number of times each test case was executed by the inner loop;
- `MCnt` is the number of times that each test case has been iterated `NCnt` times. These repetitions are carried out by the outer loop as explained above;
- `Msg` indicates whether the test is actually valid (`OK`) or not (`KO`).

2.3 Footprint And Stack Measurement

ERB first features static memory measurements. For each test case, the overall size of the code and data section is computed for the support packages binary files, the test case binary file (`$test.o` file) and the linked test case executable file. The runtime footprint is computed by subtracting the size of the test case sections and the support packages sections to the overall binary file sections. This is obviously only an approximation, since some runtime code might be for instance inlined, but this proved sufficiently accurate.

Most of the tools used to determine the memory behavior after execution are analyzing heap allocations. There are indeed usually used for detection of anomalous heap use such as memory leaks. But the different techniques used to achieve this goal can be applied to stack measurement as well.

– **External monitoring:** A simulator, a virtual machine or a debugger is used to run the program; some inspection points are set, e.g. breakpoints. For example, `gnatmem` [9] and `Valgrind` [3] uses this method. For heap usage, the inspection points are the library routine used for dynamic allocation and deallocation, e.g. `malloc` and `free`. As for stack usage, the inspection points would be every stack-modifying instruction in the code. Needless to say, it has a great cost as every time a stack-modifying instruction is hit, the program is stopped; and such an event happens very often in a program execution. This method is very precise; however, our preliminary tests showed that this method was not adapted to benchmarking: hours were needed to run a simple benchmark, e.g. Dhrystone, on a fairly fast PC machine. Moreover, this solution is highly target-dependent and technology-dependent.
– **Instrumentation:** the user code is instrumented at some particular locations, known as inspection points, by the user, the compilation tool-chain or by a binary patching utility. If they are set in the user source code, many stack variations will be lost and the result will be highly imprecise, with no way to evaluate the error; setting them automatically is highly technology-dependent and target-dependent.
– **Pattern filling:** memory is filled with a known pattern, traditionally `0xdeadbeef`, before execution. After execution, the content of the memory is read to determine the areas that have been modified. This is the kind of technology GNAT's `Debug_Pools` [9] use. The memory can be filled either externally, by a debugger, or internally, by calling a support library. In our context, the second method is less technology-dependent, and should be easier to port. This method has several systematic errors, but they are limited and measurable as shown below. It is important to note that when the stack grows into the pattern zone, not all the patterns are destroyed: because of alignment constraints, allocations without actual writings, some of the pattern are always left.

For the purpose of a portable harness, only the third solution is practical and has been selected for ERB: we provide instrumentation routines for filling stacks with a given pattern, reading the stack after execution and output the results.

The systematic errors have been carefully studied. They are mainly caused by the effects on the stack of the instrumentation routines themselves:

– **Bottom offset:** The procedure used to fill the stack with a given pattern has itself a stack frame. The value of the stack pointer in this procedure is therefore different from the value before the call to the instrumentation procedure. In order to minimize this error, the user should get the address of variable defi ned on the stack and pass it to the procedure. That value will be used for indicating the bottom limit of the stack instead of the value the procedure could guess.

- **Instrumentation clobber when writing the pattern:** The procedure used to fi ll the stack with a given pattern will itself have a stack frame. Therefore, it will only be able to fi ll the stack after its own stack frame. This part of the stack will appear as used in the fi nal measure. As the user pass the value of the bottom of stack to the instrumentation to deal with the bottom offset error, and as as the instrumentation procedure knows where the pattern fi lling start on the stack, the difference between the two values is the minimum stack usage accessible through this method. If the pattern zone has been left untouched at the end of the test, it is possible to conclude that the stack usage is inferior to this minimum stack usage.
- **Instrumentation clobber when reading the pattern:** The procedure used to read the stack at the end of the execution clobbers the stack by allocating its stack frame. If this stack frame is bigger than the total stack used by the user code at this point, it will increase the measured stack size. In order to fi nd out whether such a situation actually happens, it is possible to augment this stack frame and see if it changes the measure. To do that, an additional array of is allocated in this frame. A specifi c discriminant can be used to change its size.
- **Pattern zone overflow:** If the stack grows outer than the outermost bound of the pattern zone, the outermost region modifi ed in the pattern is not the maximum value of the stack pointer at execution. At the end of the execution, the difference between the outermost memory region modifi ed in the pattern zone and the outermost bound of the pattern zone can be understood as the biggest allocation that the method could have detect, provided that there is no "Untouched allocated zone" error and no "Pattern usage in user code" error. If no object in the user code is likely to have this size, this is not likely to happen.
- **Pattern usage in user code:** The pattern can be found in the object of the user code. In this case, the address space where this object has been allocated will appear as untouched. To avoid this situation, we have chosen a pattern which is unusual in user code, namely `0xdeadbeef`. We also enforced a rule forbidding the use of this pattern in the test cases.
- **Stack overflow:** If the pattern zone does not fi t on the stack, this may override the stack space of another task and lead to some erroneous execution. To work around this, we specifi ed large enough task sizes and adapted the pattern zone size accordingly.
- **Inlined instrumentation code:** If the instrumentation code is inlined, the objects allocated by the instrumentation procedures are allocated on the caller stack frame, which is therefore augmented. to avoid this, none of the instrumentation procedures is inlined.
- **Untouched allocated zone:** The user code may allocate objects that are never modifi ed. In this case, the pattern will not be changed. Unfortunately, there is no way to detect this error. This error does not happen often, and it is most probably due to bugs in the user code, e.g. some uninitialized variable. It is most of the time harm-

less, since it influences the measure only if the untouched allocated zone happens to be located at the outermost value of the stack pointer for the whole execution.

These systematic errors are documented precisely and methods for evaluating them are provided. The instrumentation routines are also designed to limit them by having a stack frame as small as possible.

The instrumentation code is regular Ada code, so it is not technology-dependent. Only one parameter of this implementation is target-dependent and must be adapted to a each confi guration: this parameter indicates whether the stack grows up (from low addresses to high addresses) or down. It is designed as a general stack usage measurement library, so it is not only usable for benchmarking purposes but also could be useful for evaluating the stack usage of the tasks of an Ada application.

A typical usage of the memory instrumentation package would be similar to what is shown below:

```
A : Stack_Analyzer (16#DEAD_BEEF#, Proposed_Storage_Size / 2, 0);
-- This private object is used by the memory instrumentation code.

task T is
   pragma Storage_Size (Proposed_Storage_Size);
end T;

task body T is
   Bottom_Of_Stack : aliased Integer;
   -- Bottom_Of_Stack'Address will be used as an approximation of
   -- the bottom of stack. A good practise is to avoid allocating
   -- other local variables on this stack, as it would degrade
   -- the quality of this approximation.

begin
   Fill_Stack (A, To_Stack_Address (Bottom_Of_Stack));
   Some_User_Code;
   Compute_Result (A);
   Report_Result (A);
end T;
```

A typical output of the harness looks like what follows[3]:

```
+----------------------------------------------+
|      Test |  Stack |    Gap |   Fill |   Comp |
+----------------------------------------------+
| hl_m_a_02 |    664 | 261948 |    464 |    472 |
| hl_m_a_03 |    976 | 261368 |    196 |    204 |
| hl_m_a_04 |   2336 | 260052 |    240 |    248 |
| hl_m_a_05 |   1204 | 261264 |    320 |    328 |
```

[3] All figures are in bytes

```
| hl_m_a_06 |    8408 | 261996 |    8256 |    8264 |
| hl_m_a_07 |   10916 | 259488 |    8256 |    8264 |
| hl_m_a_08 |    8352 | 262060 |    8264 |    8272 |
| hl_m_a_09 |   41904 | 260524 |   40280 |   40288 |
+-----------------------------------------------+
```

– `Test` is the name of the test being done. If there are several tasks in the test, there is one line per task;
– `Stack` is an evaluation of the stack consumption;
– `Gap` is the size of the space fi lled by the pattern that is beyond the last modifi ed area;
– `Fill` is the size of the `Fill_Stack` stack frame;
– `Comp` is the size of the `Compute_Result` stack frame;

A consequence of the above defi nitions is the following relation:
$Fill + Pattern\_Size = Gap + Stack$, where `Pattern_Size` is the size of the memory space fi lled with `0xdeadbeef`.

2.4 Test Base

In order to provide a signifi cant test base on a cost-effective basis, we decided to leverage on the extensive test bases from existing Ada benchmarks and test suites, namely the ACES project, the PIWG project, the ACATS and the ORK project Ravenscar test suite.

A list of tests was agreed upon with the Agency, specifying for each test whether execution time and/or memory measurements were required. The list comprises 174 Ada timing tests, 92 Ada memory tests, 92 RTEMS timing tests and 50 RTEMS memory tests, divided in twelve categories:

– high level algorithms
– arithmetic tests
– data storage
– data structure
– tasking and protected objects
– exception handling
– runtime checks
– iterations
– procedure and function calls
– generics
– object oriented
– miscellaneous

Those categories globally match the traditional categories used in the PIWG and ACES projects.

Adapting the tests to the project is a challenging task, since they come from different backgrounds. The tests coming from the PIWG and ACES projects are easy to port, because they rely only on a limited number of support packages. We enforced a simple porting policy: support package required for several tests were merged into our own support packages, while support packages required for a single, or a very small set of test cases was merged into the test case itself.

Additionally, we used the `gnatpp` tool to give to all the test cases a common style, since it was not homogeneous: not only the ACES and PIWG have different coding styles, but also the ACES source code is far from uniform. `gnatpp`, the GNAT pretty printer, can be used to control the indentation layout and the casing of some Ada code. The same thing was done for C tests, this time using the `indent` GNU tool, which basically does the same thing for C code.

Rather than trying to port general-purpose Ada 95 tasking tests from other benchmarks, it appeared soon that the best way to go for the tasking and protected object part was to adapt the Open Ravenscar project [12] test cases. The ORK test suite, which was initially written by EADS-CASA for the Open Ravenscar project , aims at checking compliance of given Ravenscar implementations to the profi le. It was the best starting point for writing a comprehensive Ravenscar benchmark because it uses typical Ravenscar constructs.

Porting those tests was not a trivial work though. Those tests are validation tests and not performance tests, which caused some adaptation effort. Paradoxically, the simple tests caused more trouble: because of the fi ne-grain, user code-oriented approach of the dual loop method, the measurement of execution time for creating a null task, for instance, were beyond the measurement capabilities of the harness. In that particular case, most of the task creation code is in the elaboration code coming from the runtime, which cannot possibly be instrumented. In order to workaround this limit, we decided to implement the complementary "run-all" and "run-it-once" measurement strategies.

In order to validate the set of tests, a simple strategy of consistency checking was adopted. Since the whole point of this project is making comparison between Ravenscar tool-chains, the emphasis was not put on ensuring correctness of the test cases, but more making sure that a given test case behaves consistently on all target environments. For instance, if a test raises a `Program_Error` exception with GNAT Pro, we made sure that it does the same thing at the same place with the other target environments. Additionally, reusing many test cases from existing, well-known benchmarks provides a guaranty that most of the tests are correct and have a meaningful target.

3 Conclusion

The European Space Agency and AdaCore agreed to make available upon completion of the project both the test harness and the test suite to the Ada real-time community under the terms of the GNU General Public License. As part of the original contract, an advocacy effort, undertaken by Tullio Vardanega from the University of Padua, will be carried out to increase project awareness among the community. Concrete actions that might be done possibly include advertisement in specialized newsgroups, standardization bodies, such as the IRTW group or the ARG ISO/WG9 subgroup, as well as traditional conferences.

The fi rst audience targeted by ERB is the Ada vendors community. Running the harness with other Ada tool-chains is a good way to compare them and fi nd areas where a given technology could be improved with respect to competition. Ada vendors could therefore use the harness to fi nd out the strength and weaknesses of their technology, and fi x them. They are also welcome to write test exercising particular languages features or stressing the implementation in a particular fashion. The next revision of the Ada standard is an excellent opportunity for extending the current test base so that it covers the newest language features. In any event, AdaCore will take into account comments issued by Ada vendors and feed them back into the harness,

Another example of what Ada vendors could do is also provided by AdaCore, which plans to use ERB internally to monitor the evolution of GNAT Pro and verify that performance changes match development expectations.

The second target audience we envision for the ERB project is the Ada Space community. The ERB project can be used to defi ne guidelines for avoiding the use of languages features that cost too much for a particular application. Note that the fi ndings about which features are to be avoided can change from one project to another, according to particular constraints or requirements.

If the interest we have in this project is shared by the community, AdaCore will be glad to provide some support for the long-term life of the ERB project, for example by making it, as well as third-parties contributions, a part of its effort to support libre software.

Acknowledgments

This work would not have been possible without the kind interest of Morten Rytter Nielsen, who initiated the project at the European Space Agency and supervised it as the Agency Technical Offi cer.

Many people at AdaCore have been involved into this project to some extent. This includes in particular Cyrille Comar, Franco Gasperoni, Olivier Hainque, Nicolas Roche, Arnaud Charlet. All our thanks to them for their support, suggestions and reviews.

We would also like to thank the team of the Real-Time group at the School of Telecommunication of the University of Madrid, which undertook the work of porting the test suite to RTEMS. Our gratitude goes in particular to Juan-Antonio de la Puente for the management work he did throughout the project and to Juan Zamorano and Santiago Palomino for their actual technical work and expertise.

Finally, we would like to thank Tullio Vardanega, from the Department of Applied Mathematics of the University of Padua, for his support and expertise.

References

1. *RTEMS -the Real-Time Operating System for Multiprocessor Systems*, v4.6.1 edition. Available at `http://www.rtems.com/`.
2. *The Standard Performance Evaluation Corporation (SPEC)*, spec cpu 2004 edition. Information available at `http://www.spechbench.org/`.
3. *Valgrind - a GPL'd system for debugging and profiling x86-Linux programs*. Information available at `http://www.valgrind.kde.org/`.
4. *VxWorks*, v5.x edition. Information available at `http://www.windriver.com/`.

5. *Ada Compiler Evaluation system Reader's Guide for Version 2.1*, February 1996.
6. *SIS - SPARC Instruction Set Simulator*, v3.0.5 edition, 1999. Available at `ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/doc/sis-3.0.5.pdf`.
7. Ravenscar profile for high-integrity systems, March 2003. available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT`.
8. Ada Core Technologies. *GNAT Reference Manual*, 5.02a1 edition.
9. Ada Core Technologies. *GNAT User's guide*, 5.02a1 edition.
10. Aonix. *ObjectAda Real-Time Raven.* Information available at `http://www.aonix.com/objectada_raven.html`.
11. Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. Technical Report YCS 348, University of York, 2003.
12. Juan A. de la Puente, José Ruiz, Juan Zamorano, Jes us Gonz alez Barahona, Ram on Fern andez Marina, and Miguel anguel Ajo. Open ravenscar real time kernel v2.2b. Technical report, European Space Agency, Technical University of Madrid (UPM), University of York, 2003. Available at `http://wwww.dit.upm.es/ork/`.
13. Juan A. de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In Hubert B. Keller and Erhard Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.
14. LLC EEMBC Certification Laboratories. The eembc benchmark. Information available at `http://ebenchmarks.com/`.
15. ESA/ESTEC. Standard Test Suite Elaboration and Application for Compilers and Kernels-Statement of Work. TOS-EME/02-85/MRN, October 2002.
16. G. Chen et al. Pennbench: a benchmark suite for embedded java. In *5th Workshop on Workload Characterization (WWC5)*, 2002.
17. ACT Europe. Standard Test Suite Elaboration and Application for Compilers and Kernels Requirement Baseline Document.
18. ACT Europe. Proposal for Standard Test Suite Elaboration and Application for Compilers and Kernels. December 2003.
19. Jiri Gaisler. *TSIM Simulator User's Manual.* Gaisler Research, version 1.2.3 edition, August 2003. Available at `http://www.gaisler.com`.
20. Matthew R. Guthaus, Jeffrey S. Ringenberg, and Dan Ernst Todd M. Austin Trevor Mudge Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
21. Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
22. Performance Issues Working Group (PIWG). The piwg benchmark. Available at `http://unicoi.kennesaw.edu/ase/support/cardcatx/piwg.htm`, 1993.
23. S.T. Taft, R.A. Duff, and E. Ploederer. *Consolidated Ada Reference Manual.* Number LNCS 2219 in ANSI/ISO/IEC-8652:1995. Springer-Verlag, 1995.
24. TEMIC. *SPARC V7 Instruction Set Manual*, 1996.
25. XGC. *ERC32 Ada*, 1.7 edition. Information available at `http://www.xgc.com/erc32-ada/erc32ada1.htm`.