Mission-Critical On-Board Software Using the Ada 95 Ravenscar Profile*

José F. Ruiz

AdaCore
8 rue de Milan
75009 Paris, France
Phone: +33 (0)1 49 70 67 16
Fax: +33 (0)1 49 70 05 52

ruiz@adacore.com

Abstract

This paper describes how GNAT Pro for ERC32 and the Ravenscar profile are suitable for designing and implementing complex on-board software using high-level tasking facilities. The static and simple tasking model defined by the Ravenscar profile allows for a streamlined implementation of the run-time system directly on top of bare machines. The reduced size and complexity of the run time, together with its configurability, makes it suitable for mission-critical space applications in which certification or reduced footprint is needed. Software reliability and predictability is also increased by excluding non-deterministic and non analysable tasking features. Product validation has been achieved by means of a comprehensive test suite intended to check compliance with the Ravenscar profile and Ada standards, and correct behaviour of specialised features and supplemental tools. Code coverage analysis is also part of the validation campaign, with the goal of achieving 100% statement coverage.

1 Introduction

Ada [2] is the language of choice for many critical systems due to its careful design and the existence of clear guidelines for building high integrity systems [19]. As the functionality and complexity of on-board software increases, more attention is being devoted to high level, abstract development methods. The Ada tasking model provides concurrency as a means of decoupling application activities, and hence making software easier to design and test [29].

However, tasking capabilities have been considered as too complex for mission critical systems because accurate timing analysis is difficult to achieve. Advances in real-time systems timing analysis methods have paved the way to reliable tasking in Ada. The Ravenscar profile [8, 4, 3] is a subset of Ada 95 tasking that provides the basis for the implementation of deterministic and time analysable applications on top of a streamlined run-time system.

This paper describes how the Ravenscar profile and the GNAT Pro cross-development system can be used for designing and developing mission-critical space applications in which certification or reduced footprint is needed.

2 The Ravenscar profile

The Ravenscar profile [8, 4, 3] defines a subset of the tasking features of Ada which is amenable to static analysis for high integrity system certification, and that can be supported by a small, reliable run-time system. This profile is founded on state-of-the-art, deterministic concurrency constructs that are adequate for constructing most types of real-time software [9]. Major benefits of this model are:

- Improved memory and execution time efficiency, by removing high overhead or complex features.

[*] This work has been funded by ESA/ESTEC contract No.17360/03/NL/JA and carried out in cooperation between AdaCore and the Technical University of Madrid.

- Increased reliability and predictability, by removing non-deterministic and non analysable features.
- Reduced certification cost by removing complex features of the language, thus simplifying the generation of proof of predictability, reliability, and safety.

The tasking model defined by the profile includes a fixed set of library level tasks and protected types and objects, a maximum of one protected entry per protected object with a simple boolean barrier and no entry queues for synchronisation, a real-time clock, absolute delays, deterministic fixed-priority preemptive scheduling with ceiling locking access to protected objects, and protected procedure interrupt handlers, as well as some other features. Other features, such as dynamic tasks and protected objects, task entries, dynamic priorities, select statements, asynchronous transfer of control, relative delays, or calendar clock, are forbidden. The profile is based on a computation model with the following features:

- A single processor.
- A fixed number of tasks.
- A single invocation event per task (either time-triggered or event-triggered tasks).
- Task interaction only by means of shared data (protected objects) with mutually exclusive access.

The Ravenscar profile is well adapted to space applications. It defines a computation model similar to the one proposed by Vardanega [28], which is based on the HRT-HOOD method [10]. The profile allows implementing space on-board systems using the tasking facilities provided by Ada, restricted so as to ensure that the system can be analysed for accurate timing and safety requirements. Preliminary experience confirms the validity of this approach for on-board software development [27].

The compiler and run time have been developed to be fully compliant with the latest definition of the Ravenscar profile [4, 3], so that it will be compliant with the forthcoming ISO standard revision of the Ada language.

3 Toolset overview

GNAT Pro for ERC32 is a flexible cross-development system supporting the Ravenscar tasking model on top of bare ERC32 computers. ERC32 [13, 5] is a highly integrated, high-performance 32-bit RISC embedded processor implementing the SPARC architecture V7 specification. It has been developed with the support of the European Space Agency (ESA) as the current standard processor for spacecraft on-board computer systems.

In addition to a large number of compiler features intended to detect violations of the Ravenscar profile limitations (and any other imposed restrictions) at compile time, the key element is the provision of a restricted Ada run time that takes full advantage of the Ravenscar profile restrictions [4]. Additional restrictions on the Ada subset to be used can be enforced in order to properly support the development of high integrity systems [19]. The purpose of such restrictions is to enable a wide range of static analysis techniques, including schedulability analysis, to be performed on the software for validation purposes.

The developed Ada run times take full advantage of the largely enhanced modularity introduced in GNAT Pro recently. Key to achieving this goal is the fully configurable and customisable run-time library, which allows for limiting the run-time library just to those units required for the application.

The cross-development environment provides a full-featured visual programming environment that covers the whole development cycle (language-oriented editing, compiling, binding, linking, loading, graphical tasking-aware debugging).

Quality assurance is a key part of this project, and it has been achieved by following the ECSS-E40B standard [14], and by extensive testing and source coverage analysis. A comprehensive test suite has been developed for ensuring that the compiler and run times are compliant with the Ravenscar profile and Ada standards, and for endorsing the correct behaviour of specialised features (such as the last-chance exception handler mechanism) and supplemental tools (such as the debugger). These tests have been supplemented with a part of the Ada Conformity Assessment Test Suite (ACATS) [1] which corresponds to those tests compliant to the Ravenscar profile and the limitations of this bare board environment (more than 1200 tests). Source coverage analysis is also part of the validation campaign, with the objective of achieving 100% statement coverage of the *High Integrity Ravenscar* run time.

The work described is this paper builds on some of the results of previous ESA projects which resulted in the development of UPM's *Open Ravenscar Kernel* (ORK) [11, 31], an open-source development aimed at demonstrating the feasibility of a Ravenscar-compliant Ada run time on top of a bare ERC32.

4 The high integrity approach

The high integrity edition of the GNAT Pro compiler is intended to reduce costs and risks in developing and certifying systems that have to meet safety standards, such as DO-178B [21], DEF Stan 00-55 [20], and IEC 61508 [17].

Like all general-purpose languages, only a subset of the full language is appropriate for mission-critical applications because the full language includes facilities that are difficult to analyse and verify to the degree required (see the ISO 15942 technical report [19] for more details on the assessment of Ada features).

The centerpiece of this approach is the "configurable run time" capability. Application developers and system integrators can together define an Ada subset that closely fits the needs of the projects, thus limiting the cost of certification of the run time. Run-time subsets are defined by using three different mechanisms:

- Setting parameters in the *System* package.
- Including only a subset of available run-time system units.
- Using *pragma Restrictions*.

The compiler will then flag and reject the use of constructs that are not supported by the defined subset. Developers may use presupplied implementations of units of interest, or may develop their own alternatives. This approach gives great control over the scope of certification activities when developing in Ada.

A profile is a compiler-enforced Ada language subset with a corresponding (possibly empty) run-time library. Thus selecting a profile has two effects:

- The compiler will reject any source file that uses features outside the chosen subset.
- The run-time library (if any) bound with the program will contain support only for the features in the chosen subset.

This way, users can configure a tailored library reflecting exactly the set of features that are used.

There may be cases where the use of these configuration mechanisms is not expressive enough, and there is a need for a convenient way to define and enforce code guidelines fitting mission-critical requirements. This is the case, for example, if we want to detect subprograms that modify non-local data. A tool is being developed for defining and statically detecting features and constructions deemed to be unsuitable for mission-critical systems.

Traceability from Ada source code to object code is facilitated by giving access to different intermediate formats internally generated by the compiler. From the initial source code the compiler generates a simplified code, which is low level Ada pseudo-code (target independent) that expands complex constructs into a sequence of simpler data and code (including run-time calls). This code is then compiled into assembler code, which is later transformed into object code. The availability of these intermediate representations helps certification of object code by reducing the semantic gap between different representations. Additionally, representation information for declared types and objects is also accessible.

Full Safety and Security Annex support [2, H] is provided, including capabilities for detecting uninitialised variables [12], by means of compiler warnings and run-time errors (using *pragma Normalize_Scalars* and some additional validity checking levels that can be selected by the users).

There are also plans for developing a stack analysis tool that will be able to determine worst case stack requirements for the different tasks. It will be done by analysing the full call graph and each subprogram's stack.

The forthcoming Ada 2005 standard has been enhanced to better address the needs of the real-time and high-integrity communities. This new standard introduces new restriction identifiers that can be used to define highly efficient, simple, and predictable run-time profiles. Among others, this language revision will standardise the Ravenscar profile, and will include execution time clocks and timers. Object-Oriented Programming (OOP) is also being largely enhanced, and Ada 2005 will make certification of OOP easier by means of including the recommendations made by the *Handbook for Object-Oriented Technology in Aviation* [15]. Many Ada 2005 features are already supported by GNAT Pro, and there are plans to continue implementing the new features as the proposals are stabilised and approved for the Ada 2005 revision.

5 Run-time profiles

The ERC32 development environment includes three specific instantiations of the configurable run-time library, each corresponding to a particular set of run-time Ada features.

5.1 The Zero FootPrint run time

The *Zero FootPrint* run time guarantees that the generated object modules contain no references to the GNAT Pro run-time library. This allows the construction of a standalone program that has no code other than that corresponding to the original source code (apart from the elaboration routine generated by the binder). The elaboration routine generated by the binder also avoids any reference to run-time routines or data. This run time is designed to reduce the cost of meeting safety certification standards for applications written in Ada. In addition, this profile is compatible with SPARK [7].

Although limited in terms of dynamic Ada semantics, this run time fully support static Ada constructs such as generic templates and child units, tagged types (at library level), and other object-oriented programming features. Users can also further restrict certain Ada features (such as dynamic dispatching, allocators, unconstrained objects, implicit conditionals and loops) through appropriate *pragma Restrictions*. Exception propagation is not allowed, but exception declarations and raise statements are still permitted. No exception handlers are permitted, and a user-defined last chance exception handler is executed for any exception occurrence.

5.2 The High Integrity Ravenscar run time

The *High Integrity Ravenscar* run time has been designed to accommodate certification requirements for concurrent high-integrity (safety-critical) real-time systems. It offers a multitasking programming environment (compliant with the Ravenscar profile) with maximum performances, targeted at applications aiming at certification for safety-critical use or very small footprints.

This run-time profile is a superset of the *Zero Footprint Profile* (see Section 5.1), supplementing it with the Ada tasking features permitted in the Ravenscar Profile:

- Task type and task object declarations in library-level packages.
- Protected type and protected object declarations in library-level packages.
- Absolute delay (delay until) statements.
- References to package Ada.Real_Time.
- Count attribute (but not in barrier expression).
- References to package Ada.Task_Identification (but no Abort_Task).
- Protected procedures as interrupt handlers.
- FIFO_Within_Priority dispatching policy.
- Ceiling_Locking locking policy.

Apart from these tasking capabilities, no other feature has been added from the sequential part of the Ada 95 language. This way this run-time profile offers maximum performances, both in terms of footprint and execution time, at the expense of restrictions in the availability of sequential Ada functionalities. Section 5.3 defines a run-time profile which is much less restrictive in terms of available Ada functionalities. The reduced complexity of this run time allowed 100% statement coverage when running the validation test suite developed for this run time.

The part of this run time which is written in Ada is made up by less than 1400 logical single lines of code, plus around 400 lines of assembly code. The resulting footprint of a simple tasking program compiled using this run time (including both data and code, but excluding stacks and the trap table) is around 10KB.

5.3 The Extended Ravenscar run time

The *Extended Ravenscar* run time offers support for a larger subset of Ada 95, under the restrictions of the Ravenscar profile and the hardware constraints. This profile makes software development easier (debugging, assertions, text output, stack checking, stack tracebacks, etc.), at the expenses of a larger footprint and an increased complexity in the run time.

This run-time profile is a superset of the *High Integrity Ravenscar* defined in Section 5.2, that supplements that profile with those sequential Ada language constructs that desirable for building programs to be executed on a ERC32 board, while keeping a reasonable level of complexity.

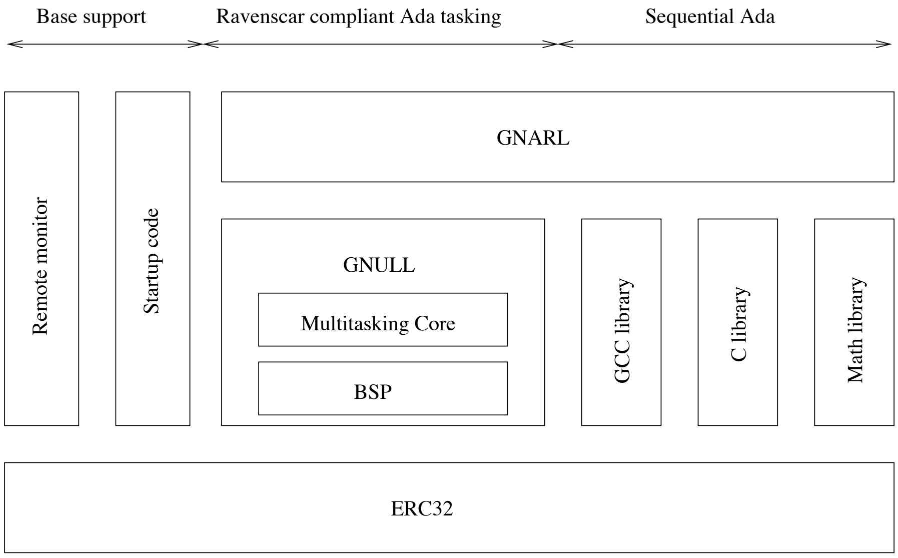

Figure 1: Run-time architecture

6 Run-time architecture

The run time system is made up by several libraries (see Figure 1) that implement functionalities required by features not otherwise generated directly by the compiler. The complexity of the run time basically depends on the features supported.

A compact and efficient run time has been designed to take full advantage of the Ravenscar Profile restrictions, which is substantially different from the run time used when no such restrictions are in effect. The Ravenscar run time provides simplified, more efficient versions for the set of tasking and synchronisation operations.

The Ravenscar run time has been carefully designed to isolate target dependencies by means of a layered architecture. There is a target independent layer, called GNU Ada Run-Time Library (GNARL), which provides the interface that is known by the compiler. The compiler translates high-level Ada constructions into a set of calls to this interface.

The part of the run time that depends on the particular machine and operating system is known as GNU Low-Level Library (GNULL), which provides a target independent interface. GNULL is some glue code that translates this generic interface into calls to the operating system interface, thus facilitating portability. On bare board targets (such as the ERC32 one), GNULL is a full implementation of this interface.

Hence, retargeting the run time to a different operating system is a matter of mapping the GNULL interface (roughly a dozen primitives for creating threads, suspending them, etc.) into the equivalent operations provided by the operating system. Retargeting the run time to a different bare board system requires reimplementing the GNULL layer on top of the Board Support Package (BSP).

6.1 Static tasking model

The implementation takes full advantage of the static Ravenscar tasking model, in which only library level non-terminating tasks are allowed.

First, the complete set of tasks and associated parameters (such as their stack sizes) are identified and defined at compile time, so that the required data structures (task descriptors and stacks) can be statically created by the compiler as global data. Hence, memory requirements can be determined at link time (linking will fail if available memory is not enough) and there is no need for using dynamic memory at run time.

In addition, task creation and activation is very simple and deterministic: the environment task (as part of its elaboration) creates all the tasks in the system, and once that is done all tasks are then activated and executed concurrently, being scheduled according to their priority.

Finally, only library level non-terminating tasks are allowed, so that there is no need for code for completing or finalising tasks.

6.2 Simple protected object operations

Protected object operations can be easily implemented taking advantage of the restrictions imposed by the Ravenscar profile:

- No asynchronous operations. There are no abort statements and no timed or conditional entry calls.
- Simple creation and finalisation of protected objects. Protected objects are only allowed at library level, and allocators are not allowed for protected types or types containing protected type components.
- Simple management of entry queues. Only one entry is allowed per protected object, with at most one task waiting on a closed entry barrier. In addition, requeues are not allowed.
- Simple priority handling. Dynamic priorities are not allowed.
- Simple locking operations. On a single processor implementation (such as the ERC32), the ceiling priority rules and the strictly preemptive priority scheduling policy guarantee that protected objects are always available when any task tries to use them [18, 23] (otherwise there would be another task executing at a higher priority), and hence entering/exiting to/from the protected object can simply be done by just increasing/decreasing task's priorities.

Operations related to protected objects without entries are implemented in an even simpler manner because there is no need to check whether there is any task waiting, no need to reevaluate barriers, no need to service entry queues, etc.

In addition, efficient execution of queued protected entries is achieved by implementing what is called the proxy model [16] for protected entry execution. At the end of the execution of any protected procedure (that may change the state of the barriers), if there is a task waiting on the protected object's entry, then the barrier is evaluated, and if needed, the entry is executed by the task that opened the barrier on behalf of the queued task. It enhances efficiency by avoiding unnecessary context switches.

6.3 Exception support

The Ravenscar profile does not place any explicit limit on the features of sequential Ada, and therefore it does not restrict the use of exceptions (in fact, some exception support is required by the Ravenscar profile [22]). Therefore, several schemes are defined for supporting exceptions, providing different levels of functionality and complexity.

The simplest exception scheme supported by the GNAT Pro run time is the "No Exceptions" one, that is called the "exclusion strategy" in [19]. Raise statements and exception handlers are not allowed, and no language-defined run-time checks are generated. Hence, program will become erroneous if a run-time exception does occur, so that the absence of erroneous states usually leading to the raising of an exception must be demonstrated.

The second choice corresponds to the "No Exception Handlers" mechanism (called "belt-and-braces" strategy in [19]). It seeks to avoid dependency on the exception mechanism, but recognises that a predefined exception may nevertheless occur for some unforeseen reason. Exception propagation is not allowed, but exception declarations and raise statements are still permitted. No handlers are permitted; a user-defined last chance exception handler (which cannot resume task execution) is introduced at the outermost scope level, and hence no run-time support is needed. If run-time checking is enabled, then it is possible for the predefined exceptions Constraint_Error, Program_Error, or Storage_Error to be raised at run time.

A third exception handling mechanism is implemented in the *extended Ravenscar Profile*, supporting the full semantics of Ada 83 exceptions; Ada 95 enhancements are not included. This run-time system supports propagation of exceptions and handlers for multiple tasks. The run-time library provided by this profile supports also limited Ada 95 exception occurrences, and Ada.Exceptions.Exception_Name. Mapping of the usual traps for hardware exceptions to Ada exceptions is also done.

The implementation of a forth alternative exception handling mechanism is being considered, supporting the "containment" strategy defined in [19], that would authorise exception handling close to the raising location. When an exception is raised, the exception handler is executed if it is located in any of the enclosing syntactic scopes up to the inner-most subprogram scope. In other words, exceptions are never propagated outside the subprogram where they were raised. Every exception not being handled within its inner-most subprogram scope forces the execution of the last chance handler. No run-time support is needed for exception propagation, so that there is no drawback either in efficiency nor in complexity of the run time.

7 Multitasking core

The Ravenscar profile is designed to be easily supported with a small run time. Within the framework of this project we have also designed and implemented a simple Ravenscar compliant multitasking core that is in charge of task scheduling, dispatching, and synchronisation, interrupt management, and timing services (time-keeping and delays). It implements a preemptive priority scheduling policy with ceiling locking and 256 priority levels (although this number can be easily reconfigured).

It has been written in Ada (except for some low-level code written in assembler to implement context switches and trap handling). A reduced, simple, and safe subset of Ada, following the recommendations made by the ISO 15942 technical report [19], has been used, including the following restrictions (among others):

- No exceptions are explicitly raised or handled.
- No object oriented functionality.
- No controlled types.
- No discriminated records.
- No packed arrays.
- No unconstrained objects.
- No 64-bit operations [1].
- No allocators or unchecked deallocation.
- No storage pools.
- No array and record assignments.

In order to enhance portability, it has been designed a Board Support Package (BSP) layer, giving access to key hardware dependent services, that minimises and isolates specific machine dependencies. It is made up by a few assembly files and a limited and identified set of Ada packages.

7.1 Timing services

The implementation of timing services is both accurate and efficient, providing low granularity (limited only by the oscillator) time measurements and delays together with a low overhead operation, by means of using two different hardware timers [32].

The ERC32 hardware provides two 32-bit timers (a very common arrangement on 32-bit boards) which can be programmed in either single-shot or periodic mode [5]. We use one of them as a timestamp counter and the other as a high-resolution timer. The former provides the basis for a high resolution clock, while the latter offers the required support for precise alarm handling.

Given that the maximum timestamp count that can be stored in the hardware clock is equal to 2**32 system clock ticks (215 seconds for a 20 MHz ERC32 board), which is largely insufficient for fulfilling Real-Time Systems Annex requirements [2, D.8 par. 30] of a minimum range of 50 years, a mixed hardware-software clock has been devised.

Time is represented as a 64-bit unsigned integer number of clock ticks. The hardware clock interrupts periodically, updating the most significant part (MSP) of the clock, a 32-bit unsigned integer kept in memory, while the least significant part (LSP) of the clock is held in the hardware clock register.

The 64-bit clock value very easily and efficiently, by simply concatenating the the MSP 32-bits, stored in memory. and the value stored within the hardware counter as the LSP 32-bits. Efficiency is achieved by using 32-bit operations instead of 64-bit ones (ERC32 does not provide 64-bit hardware operations). Each half of a *Time* value (MSP and LSP) is handled separately.

An efficient high resolution timer is achieved by programming the hardware timer on demand, and not periodically.

[1] Although the type Time is a 64-bit unsigned integer, it is internally handled as two concatenated 32-bit unsigned integers.

7.2 Interrupt handling

The three major goals when designing the interrupt handling mechanisms where simplicity, efficiency, and low interrupt latency.

Simplicity and efficiency are achieved by taking advantage of the Ravenscar restrictions on a single processor system; protected procedures (together with a short prologue and epilogue) are used as low level interrupt handlers, and no other intermediate synchronisation code is required.

Thanks to the use of the ceiling locking policy, the Ravenscar profile prevents the caller from getting blocked when invoking a protected procedure. The priority of a protected object which has a procedure attached to an interrupt must be at least the hardware Interrupt_Priority of that interrupt (otherwise the program is erroneous), as it is stated in the Systems Programming Annex [2, C.3.1 par. 14].

As a result, for as long as the active priority of the running task is equal to or greater than the one of an interrupt, that interrupt will not be recognised by the processor. On the contrary, the interrupt will remain pending until the active priority of the running task becomes lower than the priority of the interrupt, and only then will the interrupt be recognised and processed.

If an interrupt is recognised, then the call to the protected procedure attached to that interrupt cannot be blocked, as the protected object cannot be in use. Otherwise the active priority of the running task would be at least equal to the priority ceiling of the protected object, which cannot be true because the interrupt was recognised.

Low interrupt latency is accomplished by allowing interrupt nesting; otherwise, interrupts would be disabled until control returns back to the interrupted task, and interrupt latency would be high since high priority interrupts would not be handled while low priority interrupts are serviced.

7.3 Context switch

The Ravenscar profile provides the basis for the implementation of deterministic and time analysable applications, but to perform a precise schedulability analysis of a Ravenscar compliant application, the context switch time must be deterministic [30]. In addition, efficiency enhances system schedulability, and simplicity allows for cost-effective certification of the run time.

Efficiency has been enhanced by limiting the number of hardware registers that are saved/restored every context switch (ERC32 has 128 integer registers and 32 floating point registers accessible to the user).

The ERC32 architecture (a SPARC V7) includes the concept of register windows[5, 25]. There are two different approaches to follow for the flushing policy: either to flush all register windows or just the windows currently in use [6]. Taking advantage of the execution points at which it is not necessary to save (and also not necessary to restore) the entire state of the machine [24], the run time adopts the latter approach so as to reduce the excessive overhead of saving and restoring unused window registers. Hence, all the register windows that have been modified between two consecutive context switches are flushed on the task stack, and the new windows are loaded with the contents of the stack corresponding the the task that is about to execute.

Not only efficiency, but also the predictability of execution is a crucial concern. The worst case execution time (WCET) of the two alternative approaches is approximately the same. The adopted implementation however exhibits a better average execution time. This is of no use for timing and scheduling analysis though, which must by definition use only WCET values. Note that by automatically saving/restoring all the register windows that have been used by tasks has one interesting advantage which is predictability; before and after the context switch the state of the different register windows (as well as the current window pointer and the window invalid mask) are the same.

Another issue that has been taken into account is that not every task (and certainly not every interrupt handler) use the floating point unit. Thus, the floating point context is not flushed until necessary. The floating point state remains in the floating point registers, and does not change until another task (or interrupt handler) tries to use the floating point unit.

The ORK implementation always saves/restores the floating point registers when performing a context switch [26], leading to non-negligible performance penalties. In the case of interrupt handlers, the floating point context is saved and restored each time an interrupt is recognised, to allow user handlers the use of the floating point unit safely.

The scheme that we implement is that floating point arithmetic is disabled by default (both for tasks and interrupt handlers). Then, when getting a floating point trap the handler takes care of saving and restoring what is needed. It means that the floating point unit is disabled after every context switch, in order to avoid saving the context of the floating point unit when it is not needed.

This way, tasks and interrupt handlers that do not use the floating point unit do not have the unnecessary overhead related to saving/restoring the floating point context. Moreover, when computing the WCET, the overhead associated to saving and restoring the floating point context needs to be accounted only when a task (or interrupt handler) is about to use the floating point unit.

The interrupt latency is also reduced because interrupt handlers do not save the floating point context; only the integer context is saved in order to process interrupts.

8 Related work

This project builds on some of the results of previous ESA projects which resulted in the development of ORK [11, 31]. Among others, the *GNAT Pro for ERC32* compiler has the following advantages compared to ORK:

- The configurable run time capability allows for a fine grained selection of run-time entities.
- Duplicated and redundant code and data has been eliminated. Since the run time provides most of the functionality needed for tasking, some code and data are present both in the ORK kernel and in the GNAT Pro run time. Currently there is no separate kernel but a complete Ada run-time system with the needed information stored at the required level.
- Static creation of task descriptors and stacks (see Section 6.1). The compiler has been modified so that all tasking related data is created at compile time, removing the need for dynamic memory at run time.
- Task creation and activation has been largely simplified by means of adopting the Ravenscar profile restrictions.
- Several restricted exception models (see Section 6.3) are currently supported offering a wide range of choices which are with the recommendations made by the ISO 15942 [19] technical report.
- More efficient and deterministic context switches and interrupt handling. This part of the BSP has been redesigned in order to attain the simplicity and determinism required by high integrity real-time applications.
- ORK is based on a very old GNAT version (3.13), and there have been a lot of features added since then, such as a full-featured software development environment, a more efficient back-end code generator, etc.
- The validation test suite has been largely increased, including code coverage analysis (with the objective of achieving 100% statement coverage).
- Professional support and online consulting for Ada software development.

Just to give an idea of the simplification attained, it can be said that the ORK kernel is made up by around 1500 lines of Ada code (plus around 500 assembly lines), while the GNAT Pro equivalent functionality is currently implemented with around 1000 lines of Ada code (and less than 400 of assembly). Its simplicity has allowed us to achieve 100% statement coverage of the Ada run time.

Additionally, comparing the size of a simple tasking program (including both data and code, but excluding stacks and the trap table), the resulting footprint with *GNAT Pro for ERC32* (using the high integrity Ravenscar run time) is around 10KB, while the same executable compiled with ORK has a footprint of around 175KB.

9 Conclusions and future work

Ada is very well suite for embedded and real-time systems, and as complexity of on-board software increases, the high-level abstraction provided by the Ada language makes software easier to design and test. The forthcoming Ada 2005 standard has been enhanced to even better address the needs of the real-time and high-integrity communities, including the Ravenscar profile and OOP.

The Ravenscar profile defines an Ada subset that excludes non-deterministic and non analysable tasking features, and removes features with a high overhead or complexity. It allows for reduced and certifiable tasking run times (10KB footprint for a simple tasking program), and for achieving full statement coverage of the complete Ada run-time system supporting Ravenscar compliant tasking functionality.

The fully configurable and customisable run-time library allows for a fine-grained selection of run-time features so that the footprint and complexity of the run time can be limited. This approach gives great control of the scope of certification activities, allowing for a cost-effective use in safety-critical applications where evidences of predictability, reliability, and safety must be generated. Additionally, full source code is included.

The run time has been carefully designed to isolate target dependencies, allowing its portability to both bare board machine and real-time operating systems. We have plans for porting this work to other targets.

GNAT Pro for ERC32 is a flexible solution for large, safety-critical systems using the Ravenscar profile, allowing for developing multitasking systems for mission-critical space applications with safety requirements.

Acknowledgements

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 Officer.

Many people at AdaCore have been involved into this project to some extent. This includes in particular Jerome Guitton and Arnaud Charlet. All my thanks to them for their support, suggestions, and reviews.

I would also like to thank the team of the Real-Time Systems Group at the Technical University of Madrid, specially to Juan Antonio de la Puente, for his help with the documentation, and Juan Zamorano, for his contributions to many design and implementation details.

Finally, I want to express my gratitude to IPL, for porting and making available *AdaTEST 95* to the *GNAT Pro for ERC32* compiler so that code coverage analysis could be achieved.

References

[1] Ada Conformity Assessment Authority. *Ada Conformity Assessment Test Suite (ACATS)*. Available at `http://www.ada-auth.org/acats.html`.

[2] *Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995*, 1995. Available from Springer-Verlag, LNCS no. 1246.

[3] ARG. New pragma and additional restriction identifiers for real-time systems. Technical report, ISO/IEC/JTC1/SC22/WG9, 2003. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00305.TXT`.

[4] ARG. Ravenscar profile for high-integrity systems. Technical report, ISO/IEC/JTC1/SC22/WG9, 2003. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT`.

[5] Atmel Corporation. *TSC695F SPARC 32-bit Space Processor: User Manual*, 2003.

[6] T.P. Baker and Offer Pazy. A unified priority-based kernel for Ada. Technical report, ACM SIGAda, Ada Run-Time Environment Working Group, March 1995.

[7] John Barnes. *High Integrity Software. The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[8] Alan Burns. The Ravenscar profile. Technical report, University of York, 2002. Available at `http://www.cs.york.ac.uk/~burns/ravenscar.ps`.

[9] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003. Available at `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf`.

[10] Alan Burns and Andy Wellings. *HRT-HOOD(TM): A Structured Design Method for Hard Real-Time Ada Systems*. North-Holland, Amsterdam, 1995.

[11] Juan A. de la Puente, Juan Zamorano, José F. Ruiz, Ramón Fernández-Marina, and Rodrigo García. The design and implementation of the open ravenscar kernel. *Ada Letters*, XXI(1), March 2001.

[12] Robert Dewar, Olivier Hainque, Dirk Craeynest, and Philippe Waroquiers. Exposing uninitialized variables: Strengthening and extending run-time checks in ada. In J. Blieberger and A. Strohmeier, editors, *Reliable Software Technologies — Ada-Europe 2002*, number 2361 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[13] ESA. *32 Bit Microprocessor and Computer System Development*, 1992. Report 9848/92/NL/FM.

[14] European Cooperation for Space Standardization (ECSS). *ECCS-E-40B Space Engineering — Software*, November 2003.

[15] Federal Aviation Administration (FAA). *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, October 2004. Available at `http://www.faa.gov/certification/aircraft/av-info/software/OOT.htm`.

[16] Edward W. Giering, Frank Mueller, and Theodore P. Baker. Implementing ada 9X features using POSIX threads: Design issues. In *Proceedings of TRI-Ada 1993*, pages 214–228, 1993.

[17] IEC. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, 1998.

[18] Intermetrics. *Ada 95 Rationale: Language and Standard Libraries.*, 1995. Available from Springer-Verlag, LNCS no. 1247.

[19] ISO/IEC/JTC1/SC22/WG9. *Guidance for the use of the Ada Programming Language in High Integrity Systems*, 2000. ISO/IEC TR 15942:2000.

[20] Ministry of Defence. *DEF STAN 00-55: Requirements for Safety Related Software in Defence Equipment*, August 1997.

[21] RTCA, Inc. *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[22] José F. Ruíz, Juan A. de la Puente, Juan Zamorano, and Ramón Fernández-Marina. Exception support for the Ravenscar profile. In *Workshop on Exception Handling for a 21st Century Programming Language*, volume XXI, pages 76–79. ACM SIGAda, September 2001.

[23] H. Shen and T.P. Baker. A Linux kernel module implementation of restricted Ada tasking. *Ada Letters*, XIX(2):96–103, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[24] J.S. Snyder, D.B. Whalley, and T.P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, February 1995.

[25] Sun Microsystems Corporation. *The SPARC Architecture Manual*, 1987. Version 7.

[26] UPM. *Open Ravenscar Kernel — Software Design Document*, 1.7 edition, July 2000.

[27] T. Vardanega, G.Caspersen, and J.S. Pedersen. A case-study in the reuse of on-board embedded real-time software. In Michael González-Harbour and Juan A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe'99*, number 1622 in LNCS, pages 425–436. Springer-Verlag, 1999.

[28] Tullio Vardanega. *Development of On-Board Embedded Real-Time Systems: An Engineering Approach*. PhD thesis, TU Delft, 1998. Also available as ESA STR-260.

[29] Tullio Vardanega and Jan van Katwijk. A software process for the construction of predictable on-board embedded real-time systems. *Software Practice and Experience*, 29(3):1–32, 1999.

[30] Juan Zamorano and Juan A. de la Puente. Precise response time analysis for ravenscar kernels. In *11th International Workshop on Real-Time Ada Issues*. ACM Press, 2002.

[31] Juan Zamorano and José F. Ruiz. GNAT/ORK: An open cross-development environment for embedded Ravenscar-Ada software. In E.F. Camacho, L. Basañez, and J.A. de la Puente, editors, *15th IFAC World Congress*. Elsevier Press, 2002.

[32] Juan Zamorano, José F. Ruiz, and Juan A. de la Puente. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in Lecture Notes in Computer Science. Springer-Verlag, 2001.