Adding new features to the Open Ravenscar Kernel

Santiago Urueña  José A. Pulido  Juan Zamorano  Juan A. de la Puente

Universidad Politécnica de Madrid, Spain

Abstract

ORK is a specialized real-time kernel for high-integrity embedded systems based on the Ada Ravenscar profile. The paper is focused on the evolving requirements for a new generation of Ravenscar kernels, coming both from the evolution of the Ada language and the needs of future aerospace systems. An assessment of the changes is done, and a set of new features to be included in the next ORK version is selected. The new features are organized as an upward-compatible set of kernel configurations, which can be used in different kinds of systems.

1. Introduction

The Open Ravenscar real-time Kernel (ORK) [11, 13] is a small, reliable kernel for high-integrity embedded real-time systems which supports a simple computational model which can be analysed for temporal correctness using rate-monotonic and response-time analysis techniques [15, 6]. The ORK computational model is defined by the Ada Ravenscar profile [7, 8], and supports systems consisting of a static set of periodic and sporadic tasks communicating by means of protected shared objects. Embedded real-time applications can be built on top of ORK in Ravenscar Ada or in C, using the GNAT\(^1\) compilation system.

The current version of ORK\(^2\) has shown its value for high-integrity embedded real-time systems, after having been used in some pilot applications with very positive results (see e.g. [25]). It has been recently adopted as a basis for a professional software development system for mission-critical spacecraft embedded systems [21]. However, new requirements for supporting a wider class of embedded systems have arisen, and a proposal to update the Ada language standard, including the Ravenscar profile itself and a number of interesting new features for real-time systems, is expected to be approved soon. These changes in

\(^\ast\)Work supported by MEC, project TRECOM (TIC2002-04123), and the EC 6FP, project ASSERT (IST 4033).

\(^1\)http://www.gnu.org/software/gnat/gnat.html

\(^2\)Available at http://www.dit.upm.es/ork.
• Time keeping and absolute delays implemented with the highest possible accuracy [28].
• Storage management, restricted to linear allocation of stack space for threads at system start time.
• Interrupt handling.

The kernel is organized as a set of Ada packages which implement the above functionality (figure 1). This architecture has been designed in order to ensure portability and configurability, thus easing its maintenance and evolution.

3. Ada 2006 real-time features

3.1. The Ada 2006 revision process

The first motivation for the evolution of ORK comes from the changes in the Ada language itself. After a good amount of work by the responsible standardization bodies and the Ada community, an amendment to the language has been developed which will expectedly result in a new standard to be issued at the beginning of the next year. The amendment is aimed at improving the language in several areas, one of which is real-time systems.

The most visible enhancement is the addition of the Ravenscar profile itself to the standard [1, D13.1], thus acknowledging its relevance for the high-integrity systems domain. The main implication of this change is that all Ada compiler builders who want to offer the Ravenscar profile have to do it in a standard way, so that Ravenscar programs are portable at the compiler level. However, existing real-time kernels that support the Ravenscar profile need not be modified, as the profile definition has been stable for some years now.

In addition to making the profile part of the standard, there are other changes in the real-time area that have a potential impact on real-time kernels. Some of the changes are outside the Ravenscar definition, but they still have to be analysed in order to find possible conflicts or side effects, and to assess the possible benefits of extending the computation model so that the new features can be used in high-integrity systems. The following paragraphs include a discussion of the most important Ada 2006 changes and their impact on Ravenscar kernels.

3.2. Dynamic ceiling priorities

Dynamic task priorities is an important feature for real-time systems with multiple operating modes, which was already supported in Ada 95. However, there was some inconsistency in that the ceiling priorities of protected objects could not be dynamically changed. Although there are workarounds that enable mode changes to be effected with constant ceilings, they are error-prone and may introduce additional blocking. The proposed Ada amendment solves this inconsistency by enabling ceiling priorities to be changed at run time [1, D5.2].

Dynamic task priorities were excluded from the Ravenscar profile because they may compromise temporal predictability if used improperly [8]. The 2006 amendment sticks by this policy by keeping dynamic ceiling priorities out of the profile. This implies that no modification to the kernel is required by this change. However, if ORK would be extended to support some kind of “extended Ravenscar profile”, which has some supporters in the real-time Ada community, [14], it would be a simple matter to add support for dynamic priorities and ceilings. All that is needed is to provide kernel services for modifying the priority entries in the kernel data structures associated with tasks and protected objects, respectively.

3.3. Execution-time clocks and timers

Execution-time clocks provide a mechanism for measuring the CPU time that a task has spent since it was started [1, D14.1]. This is indeed a useful feature for high-integrity systems, as it enables execution-time budgets to be monitored and overloads to be detected. It has been shown to be compatible with the Ravenscar restrictions [12], and has been included in the new standard definition of the profile.

The Ada 2006 proposal also includes execution-time timers, which can be used to make a handler procedure to be executed whenever a specified amount of CPU time is consumed by a task [1, D14.1]. This feature enables execution-time overruns to be detected and, when applicable, error recovery to be performed. However, effective use of execution-time timers requires asynchronous transfer of control, a construct which introduces what is usually considered an unacceptable degree of indeterminacy for high-integrity systems [8]. For this reason it has been excluded from the Ravenscar profile.

A related feature is execution-time budgets for groups of tasks [1, D14.2]. The main motivation for this mechanism is to provide support for aperiodic servers [23, 24]. Group budgets have been kept out of the Ravenscar profile for similar reasons as execution-time timers.

The impact of the above changes on ORK is not the same for each of them. Execution-time clocks have to be supported, as they are part of the profile, and can be implemented in a rather straightforward way by keeping track of each task consumed CPU time on every context switch [27].

Execution-time timers and group budgets are not part of the profile, but are clearly of interest for a wider class...
of real-time systems requiring a strict control on overruns. Their implementation is more complex, as it involves using a hardware timer, which in some architectures has to be shared with the implementations of delay services. A pilot implementation has also shown that these features introduce a significant overhead on context switches and interrupt handling [27].

3.4. Scheduling policies

The current Ada standard [2] defines a preemptive priority scheduling policy, with FIFO dispatching order for tasks at the same priority, and an immediate ceiling priority access protocol for shared data encapsulated in protected objects. The Ravenscar profile mandates this scheduling policy, and forbids dynamic priority changes other than those required by the locking policy, as well as dynamic tasks and protected objects. All these restrictions define a static, analyzable task model [8].

The proposed new standard enlarges the Ada scheduling model by including additional scheduling methods in an upward-compatible way. The primary scheduling mechanism is still priorities, but different dispatching policies may be specified for different priority levels. The new dispatching policies include non-preemptive FIFO, round-robin, and earliest deadline first (EDF). Dispatching policies apply to a band of priority levels, so that different policies, e.g. EDF and preemptive FIFO may be used on the same system [1, D2].

The new scheduling policies are not part of the Ravenscar profile, but non-preemptive and EDF scheduling are clearly of interest for an “extended Ravenscar” class of applications. The impact of adding alternative scheduling policies to the kernel is comparatively high, as it involves a major modification of the thread management package (figure 1). Different dispatching procedures have to be implemented, and the one to be used at scheduling point depends on the scheduling policy which is used at the current priority level. This mechanism has a cost on context switch times which is still to be evaluated.

3.5. Timing events

Timing events enable a low-level mechanism for executing procedures at specified points of time, without using tasks or delay statements [1, D15]. This feature enables efficient implementation of short time-triggered actions, and is thus of interest for real-time embedded systems.

Timing events are supported at the library level, i.e. the set of timing events must be static. Therefore, the kernel has to be updated in order to support static timing events.

The updated Ravenscar profile definition for the new standard allows timing events, but only at the library level, i.e. the set of timing events must be static. Therefore, the kernel has to be updated in order to support static timing events.

The impact of this change on ORK is high. The approach is to use a hardware timer and an ordered queue of timing events. Delay expirations are a particular case of timing events, and thus share the same timer and event queue. The main problem is with event cancellations, which may
compromise the static, predictable nature of Ravenscar programs.

4. Requirements of future embedded systems

4.1. The ASSERT project

Technological changes in hardware and software are expected to keep the trend towards increasing flexibility and complexity in high integrity embedded systems in the forthcoming years. This trend is exemplified by the ASSERT project, which is aimed at improving the system-and-software development process for critical embedded real-time systems in the Aerospace and Transportation domains. Preliminary requirements capture for two pilot projects in the aerospace domain which has been performed in the framework of ASSERT shows that future real-time embedded systems in this domain will have significant differences in size and complexity within a single product family of related applications. This means among other things that the real-time kernels for these systems will have to be configurable and flexible enough to support the variability of future families of embedded systems.

The most important characteristics that have been identified in this analysis are:

- **Distribution.** Future embedded systems will require distributed execution on a set of possibly heterogeneous computers. Communication must be transparent with respect to the location and architecture of the communicating entities, and at least in some cases it must also be predictable in the temporal domain.

- **Criticality.** Some applications are highly critical, in the sense that their failure may lead to loss of life or mission failure. These applications are usually classified as level A or B according to DOD-178B or a similar standard, and are usually required to undergo a certification process which is also defined in the relevant standard.

- **Partitioning.** The increasing power of microprocessor hardware has led to putting together in the same computer different applications, possibly with different criticality and timing requirements. Partitions are logical spaces for the protected execution of such applications, so that storage space and processor time allocated to one application are not invaded by other applications.

- **Dependability.** In order to ensure the integrity of high-criticality applications, dependability-oriented techniques such as replication and fault containment regions may have to be used in future embedded systems.

The implications and impact of these properties on real-time kernels are analysed in the next paragraphs.

4.2. Distributed execution

Transparent communication and other functionality are appropriately handled by a middleware layer. Indeed, middleware to be used in real-time systems must exhibit a predictable temporal behaviour, which can be analysed using appropriate response-time analysis methods. PolyORB is a middleware which can be configured for different so-called *personalities*, including some real-time standards as RT-CORBA and Ada DSA. It has been adopted as one of the building blocks of a generic distributed systems architecture which is to be prototyped in the framework of the ASSERT project (see 5 below).

The impact of distribution features on the real-time kernel is mainly on the lower-level communication layers. Predictable communication requires bounded, analysable message transmission times as a basis. A number of network and communication protocols are available that have the required properties, but not all of them are appropriate for the ASSERT application domains. RT-EP and SOIS MTS have been selected for the prototype architecture. The only modification which is required from the kernel in order to support these protocols is the development of a device handler for a LAN chip and the associated RT-EP driver, which are included in the board-support package.

4.3. Partitions

In a partitioned system, computer resources are allocated to a number of partitions, in which different applications run. Each application may in turn have a number of concurrent threads. In order to ensure space and time isolation among partitions, appropriate mechanisms have to be used that prevent one application to run into another application memory space or to use processor time beyond a given budget. The same applies to other resources, such as input-output devices and communication links.

There are a number of methods that can be used to enforce inter-partition isolation. Hardware mechanisms (MMU) are customarily used to divide a physical storage space into a number of virtual memory spaces, which can be allocated to different partitions. Time partitioning can be achieved by hierarchical scheduling. A global scheduler distributes processor time among partitions, and a local scheduler determines which thread within the running partition executes at a given time. Timers of different kinds

---

4 Automated proof based System and Software Engineering for Real-Time systems, FP6 IST 4033.
can be used to detect overruns both at the local and global levels.

The ARINC 653 standard [3] for integrated modular avionics (IMA) is based on such principles. It defines an operating system interface (APEX), with two-level scheduling. Static scheduling is used for partition scheduling, in a similar way as a cyclic executive works, while thread scheduling within partitions is based on fixed-priority preemptive scheduling (FPPS). This approach leads to highly predictable, analysable timing behaviour, at the cost of an inherent lack of flexibility at the partition scheduling level.

A possible alternative is to use FPPS for scheduling partitions. This approach has the advantage of a greater flexibility and better processor utilization, although temporal analysis becomes more complex. Recent work in the framework of the FIRST project [10] provides new techniques for response time analysis of systems with hierarchical scheduling that make FPPS a real alternative to static scheduling for partitioned systems.

Another relevant issue in partitioned systems is inter-partition communication, which in the real-time case must be predictable. Communication mechanisms should be designed in such a way the integrity of the communicating partitions is not compromised.

The impact of partitioning on the kernel is deep. Although hierarchical scheduling can be implemented using priority bands and group budgets as proposed in the Ada amendment, this may not be enough for systems with a high level of criticality. A hierarchical architecture of the kernel reflecting the hierarchy of partitions and threads seems a better approach for building partitioned systems, as proposed in section 5.

4.4. Criticality and dependability

Applications running on a partitioned system may have different criticality levels. Moreover, all the software which is involved in the execution of a high-criticality application must be certified at that criticality level. This means that, in order not to have to certify all the applications to the highest level, applications must be isolated so that a failure in a low-criticality application—which may be acceptable at its level—does not compromise the execution of high-criticality applications. Indeed, partition mechanisms can be used to this purpose, including MMU hardware for spatial isolation and appropriate scheduling methods, together with temporal analysis and overrun detection mechanisms, for temporal isolation. Of course, this approach requires all the software implementing the partition mechanisms to be certified at the highest level, but in turn applications only have to be certified at their respective criticality levels. Under this approach, partitions act as failure confinement regions for the applications running within them.

The main impact that this approach has on the kernel is that it requires all the partition support software to be certified at the highest level. This includes the real-time kernel and other components which are described in section 5 below. An implication of this requirement is that the kernel and related software must be kept simple enough so that its behaviour can be shown to be predictable at all times.

5. A tailorable real-time architecture

5.1. Introduction

Not all the embedded systems of the future will require support for all the above described properties. The need for small, highly-reliable systems based on a single processor board will still exist, and the Ravenscar profile will undoubtedly be a reasonable computation model for this kind of systems. On the other hand, complex multi-partition distributed systems will also be needed, and appropriate computation models providing the required levels of predictability and dependability for such complex systems will be an issue with growing importance. As we have seen, partitions with space and time separation, hierarchical FPPS and
predictable communications and middleware offer solutions based on available technology for these systems.

Our proposal is to develop a tailorable family of real-time kernels that can give support to different kinds of systems. The baseline configuration is a Ravenscar-compliant kernel, similar to the current ORK version plus the compatible Ada 2006 extensions and communication drivers. Some possible extensions include an “extended Ravenscar profile” with dynamic priorities and ceilings, execution-time timers, and task group budgets. Support for alternative scheduling methods, such as EDF and non pre-emptive priorities, as well as priority-band scheduling, is another clear extension to the basic profile.

Supporting partitions and distributed systems is a more complex issue. A hierarchical architecture has been designed in the framework of the ASSERT project as a first step to investigate these issues, and a first prototype has been built with the aim of showing the capabilities of present day technology and learning more about the problems of complex, distributed embedded systems.

5.2. ASSERT middleware prototype architecture

The first prototype of the ASSERT middleware architecture is based on a set of components that provide support for predictable distributed execution of hard real-time systems (figure 3.) The main software components are:

- A real-time kernel that provides support to thread scheduling and other functions. In the current prototype this component is instantiated by ORK.
- A message transfer service (MTS), which is instantiated in the prototype by a SOIS-MTS package.
- A middleware layer providing support for transparent communication between distributed application components. In the prototype this layer is instantiated by PolyORB, tailored to work on SOIS-MTS and providing both RT-CORBA and Ada DSA services to applications.

The prototype works on bare PC boards linked by a dedicated Ethernet link. A sample application is built on top of it, showing a minimal but representative distributed real-time configuration.

5.3. Partitioned architecture

The above prototype architecture does not support partitions. In order to make the architecture upward compatible, the extended architecture puts all the partition management functions in a separate nano-kernel layer, leaving the upper layers unchanged or even removing it in the computer nodes that have only one partition. This approach has already been explored in other real-time domains with promising results (see e.g. [17]).

The main nano-kernel functions are:

- Memory management and spatial separation among partitions.
- Partition scheduling and temporal separation. Partitions can be scheduled according to a static execution plan, or with fixed priorities following the scheme proposed in FIRST [10]. Temporal separation can be enforced by using execution-time timers at the partition level.
- Inter-partition communication. Message-based communication provides upward compatibility with the middleware layer and location transparency at this layer.
- Virtual device handling for partition input-output.

Detailed design of the partitioned architecture is expected to be developed before the end of this year.

6. Conclusions

The need for evolution of the ORK kernel comes mainly from two sources: The Ada 2006 revision process and the increasing complexity of high-integrity embedded real-time systems. We have analysed the changes that are required, and a basic set of features have been selected to be added to the ones defined in the Ravenscar profile.

Future versions of the kernel will be tailorable, ranging from a basic Ravenscar configuration to a full-fledged kernel that is included in a partitioned, distributed architecture including also other components. A first prototype supporting predictable distribution has already been developed, and an extended prototype also supporting partitions on the same computer will soon be designed in the framework of the ASSERT project.

7 Acknowledgments

The ASSERT middleware prototype has been developed jointly with Laurent Pautet, Jerome Hugues and Khaled Barbaria from ENST, Stuart Fowler and Marek Prochazka from SciSys, and Tullio Vardanega from the University of Padua. The authors have also had many fruitful discussions with these and other members of the ASSERT project about the architectural concepts presented in this paper.
Figure 3. Distributed system architecture

Figure 4. Partitioned system architecture
References


